UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

Integrazione di un protocollo di QKD nella suite IPsec

RELATORE: Prof. Nicola Laurenti

CORRELATORE: Dott. Matteo Canale

LAUREANDO: Davide Girardi

A.A. 2010-2011

Ai miei genitori.

Indice

Sommario

Allo stato attuale la suite di protocolli IP security (IPsec) e in particolare il protocollo Internet Key exchange (IKE o IKEv2) utilizzano il metodo Diffie-Hellman per lo scambio di chiavi. Questo metodo si basa sulla difficoltà della risoluzione del problema del logaritmo discreto, e fornisce quindi una sicurezza di tipo solamente computazionale, che non dà nessuna garanzia di essere ugualmente sicuro in futuro.
In questo elaborato verrà descritto il lavoro svolto nell'A.A 2010-2011 nel tentativo di migliorare la sicurezza di IPsec valutando l'integrazione di questo con un protocollo di Quantum Key Distribution (QKD), mediante l'analisi della letteratura recente sull'argomento e l'implementazione di un modulo per la QKD da integrare con una libreria per IPsec già esistente.

Introduzione

La diffusione dei personal computer, l'introduzione delle reti locali, il proliferare dei servizi telematici, hanno segnato profondamente l'aspetto di Internet, modificando progressivamente il profilo dell'utente tipico che è sempre più orientato verso la rete.
Tutto questo, se da un verso ha comportato una grande funzionalità nelle procedure del mondo del lavoro, dall'altro ha anche costretto gli individui ad affrontare problematiche sconosciute, derivate dai nuovi rischi legati alla sicurezza delle informazioni, che hanno assunto via via sempre più importanza. A titolo d'esempio basti pensare all'internet banking e all'acquisto di beni online.
Senza sistemi per proteggere queste informazioni e la loro trasmissione nella rete, qualunque malintenzionato potrebbe accedervi.
Allo scopo di garantire un certo grado di sicurezza sono stati progettati negli anni vari protocolli di sicurezza quale ad esempio IPsec.
La quasi totalità di questi protocolli, però, è in grado di fornire dei servizi di sicurezza che si basano solamente sulle limitate capacità di elaborazione di un eventuale attaccante. Inoltre spesso non restano al passo con la continua evoluzione dei mezzi elettronici.
Nasce quindi l'esigenza di sperimentare nuovi sistemi che forniscano una sicurezza di tipo incondizionato, non dipendente quindi dalle capacità di calcolo dell'attaccante.
Nel presente elaborato, verrano illustrate nei primi due capitoli le due realtà che intendiamo integrare, IPsec e la Quantum Key Distribution (QKD) descrivendone le caratteristiche principali.
Nel terzo capitolo verrà descritto come abbiamo integrato le due realtà da un punto di vista progettuale e teorico.
Nel quarto e ultimo capitolo, verrà descritta invece la stessa cosa da un punto di vista implementativo, mostrando come è stato possibile integrare la QKD con il software OpenIKEv2.

Capitolo 1

IP security

La suite di protocolli IP security (IPsec)[1] è stata progettata dall'Internet Engineering Task Force (IETF) per fornire un framework di sicurezza al protocollo IP (Internet Protocol)[1]. La sicurezza viene quindi fornita al livello 3 (Network) della pila ISO-OSI e non dipende quindi dall'applicazione nè dal protocollo di trasporto che stiamo utilizzando. Esso può essere usato per proteggere la comunicazione fra due host, fra due gateway o fra un host e un gateway.
Considerando le varie configurazioni possibili, IPsec è in grado di fornire i seguenti servizi:

- **Confidenzialità:** Il traffico fra le due parti viene crittato, quindi solo chi possiede la chiave corretta per decrittare i dati è in grado di leggere il messaggio correttamente.

- **Integrità:** IPsec è in grado di rilevare se un pacchetto è stato modificato durante il tragitto.

- **Autenticazione:** Viene garantito che i dati provengano dal mittente corretto.

- **Anti-replay:** La presenza di un sequence number (crittato) nel payload IPsec e di altri accorgimenti ci permette di garantire che un pacchetto non venga rinviato da terzi.

IPsec opera in due modalità denominate transport e tunnel.

Transport Mode Questa modalità è utilizzata principalmente nelle comunicazioni host-to-host e prevede l'autenticazione e/o la crittazione del solo payload in ciascun datagram.

[1] http://www.ietf.org/rfc/rfc791

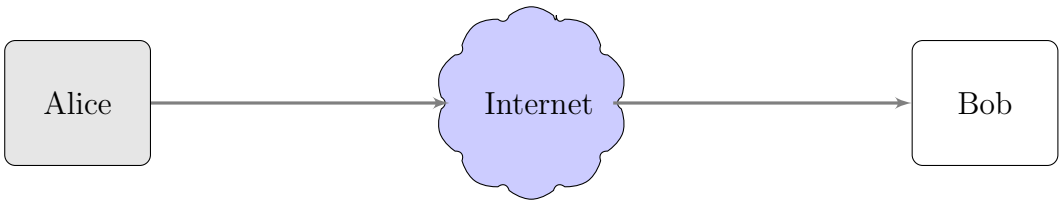


Figura 1.1: Transport mode.

Tunnel Mode La modalità tunnel viene utilizzata principalmente nelle comunicazioni gateway-to-gateway e consiste nell'autenticare e/o crittare l'intero pacchetto, incapsulandolo come payload di un nuovo pacchetto IP. Viene comunemente usato per creare reti private virtuali (VPN).

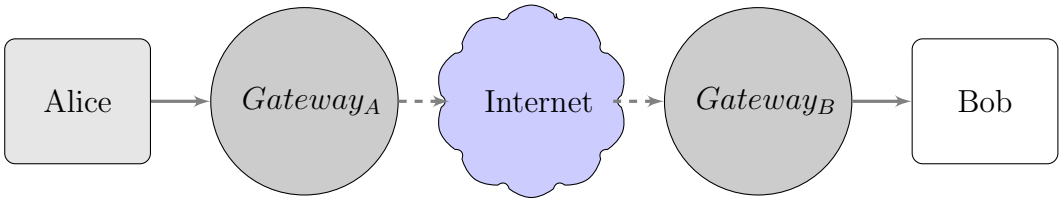


Figura 1.2: Tunnel mode.

1.1 Security Association

Il concetto di security association (SA) è uno dei componenti fondamentali che stanno alla base di IPsec. Esso consiste in un accordo negoziato fra le due parti coinvolte nella comunicazione. Ogni SA è unidirezionale, ovvero definisce le informazioni necessarie alla protezione di un solo verso della comunicazione. In altre parole, chiamando SA_{in} la security association che si occupa del traffico in entrata e SA_{out} quella che si occupa di quello in uscita, le due entità della comunicazione, tradizionalmente chiamate come Alice e Bob, creeranno le proprie SA_{in} e SA_{out} e la SA_{in} di Alice condividerà gli stessi parametri crittografici della SA_{out} di Bob. Lo stesso vale ovviamente per le altre due SA. Per memorizzare tutte le SA è previsto che ogni implementazione di IPsec realizzi un Security Association Database (SADB).
Fra i dati che compongono una SA sono presenti i seguenti campi:

- **Security Parameter Index:** valore selezionato da chi riceve una richiesta di SA per identificarla univocamente. Questo valore viene utilizzato anche per indicizzare le SA all'interno dell' SADB.

- **Sequence Number:** valore intero che viene incrementato di 1 ogniqualvolta una specifica SA viene utilizzata per proteggere un pacchetto. Sarà

compito poi del destinatario verificare questo campo in modo da prevenire eventuali replay attack.

- **Sequence Number Overflow:** flag che indica la politica da adottare in caso di overflow del sequence number. Sono previsti due casi: possiamo segnalare l'evento e non inviare altri pacchetti o resettare il contatore.

- **Anti-Replay Window:** contatore e bit-map usati per determinare se un pacchetto in entrata è stato già mandato in precedenza.

- **Dati relativi ai protocolli AH e ESP:** campi che contengono informazioni necessarie per i protocolli AH e ESP quali, ad esempio, gli algoritmi di crittazione e controllo d'integrità, le chiavi utilizzate da questi algoritmi ecc.

- **Lifetime:** intervallo di tempo dopo il quale una SA deve essere sostituita con una nuova (con un nuovo SPI) o terminata, e in aggiunta un'indicazione riguardo a quale di queste azioni si dovrebbe compiere.

- **IPsec mode:** indica se dobbiamo utilizzare la modalità tunnel o transport.

- **Dati tunnel mode:** usato solamente in modalità tunnel, indica la sorgente e la destinazione del tunnel IP.

1.2 Security Policy

Dopo aver definito *come* proteggere il traffico dati tramite le SA, dobbiamo definire *quale* traffico debba essere protetto. A questo scopo esistono le security policies (SP).
Come per le security association, anche le SP vengono memorizzate in un database, chiamato Security Policy Database (SPD). Per ogni pacchetto in invio o in ricezione sono possibili tre scelte con cui processarlo:

-PROTECT: il pacchetto viene protetto da IPsec.

-BYPASS: il pacchetto viene inviato o ricevuto senza essere protetto da IPsec.

-DISCARD: in trasmissione, il pacchetto non viene inviato; in ricezione non viene inviato ad altri nodi o agli strati superiori.

Ogni SP consiste di alcuni parametri. Quelli principali sono i seguenti:

- **Name:** nome generico con cui possiamo identificare un indirizzo IP locale o remoto.

- **Selectors:** un certo numero di gruppi di selector che corrispondono alla condizione per applicare una delle tre azioni descritte in precedenza.

  Ogni gruppo di selector contiene:

  - Uno o più indirizzi IP locali.
  - Uno o più indirizzi IP remoti.
  - il Next Layer Protocol.
  - Porta locale.
  - Porta remota

- **Processing Info:** indica quali delle tre azioni base, descritte in precedenza, intraprendere per tutti i gruppi di selector.

1.3 Authentication Header

Authentication Header (AH)[3] è uno dei protocolli contenuti nella suite IP-sec. Esso garantisce integrità dei dati, autenticazione e un servizio anti-replay opzionale ma non provvede nessun servizio di riservatezza del traffico.

1.3.1 Struttura

Nella figura sottostante è mostrata la struttura dell'header

Ottetto

1(bit 0-7)	2(bit 8-15)	3(bit 16-23)	4(bit 24-31)
Next Header	Payload Length	Reserved	
Security Parameter Index (SPI)			
Sequence Number			
Integrity check value (variabile)			

Figura 1.3: Header AH

- **Next Header:** campo di 8-bit che identifica il payload successivo a AH.
- **Payload Length:** lunghezza del payload AH in word da 32 bit, meno 2.
- **Security Parameter Index e Sequence Number:** hanno le stesse proprietà dei valori omonimi visti in precedenza.
- **Integrity check value (ICV):** valore a lunghezza variabile utilizzato per il controllo di integrità.

1.3.2 AH in tunnel e transport mode

Nella modalità Transport viene semplicemente aggiunto al pacchetto IP l'header AH, inserendolo dopo l'header IP e prima di ogni altro header di protocolli di livello superiore (TCP,UDP ecc) o di ogni altro header IPsec aggiunto precedentemente. AH opera su tutti i campi del pacchetto IP eccetto quelli variabili quali TTL, TOS ecc.
Per ulteriori dettagli si veda la figura sottostante (I campi protetti da AH sono quelli evidenziati).

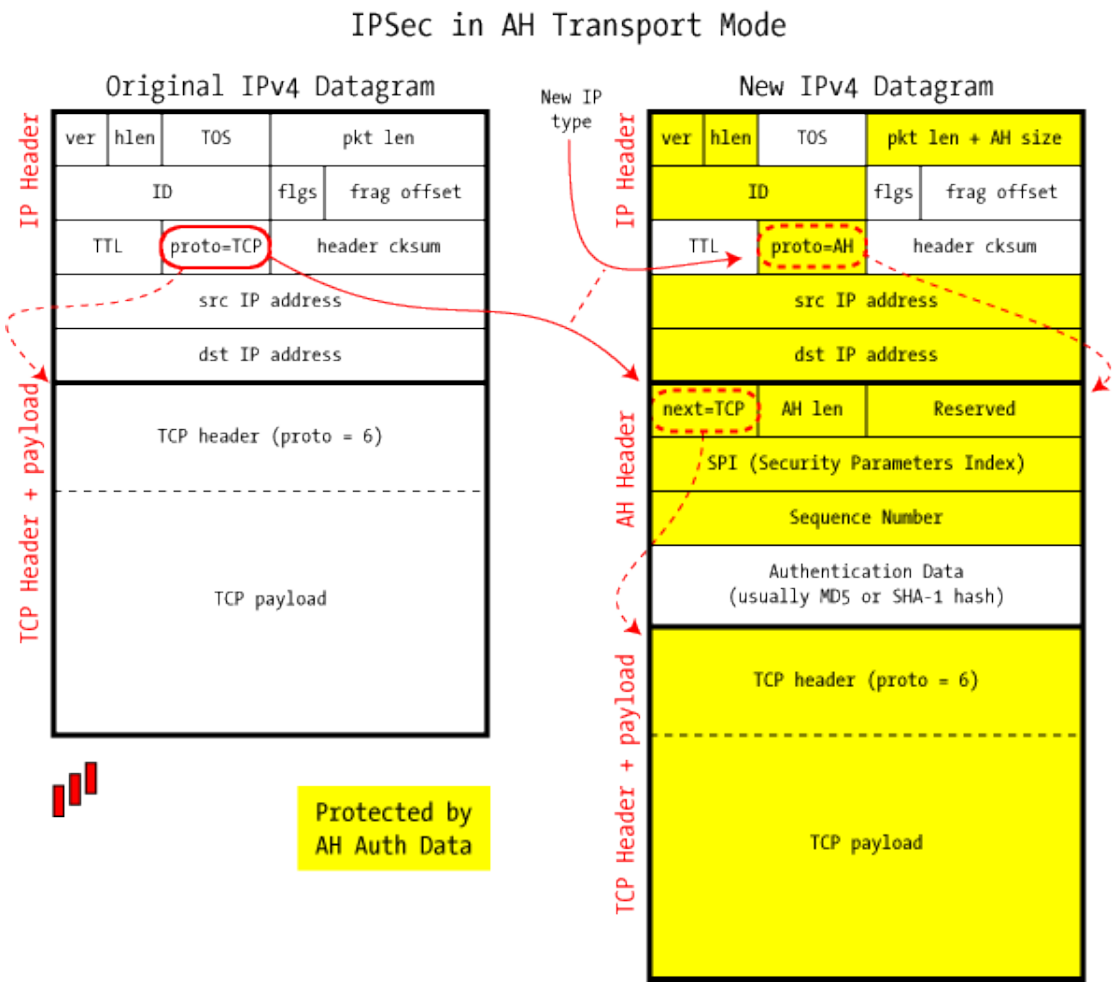


Figura 1.4: Inserimento di AH in un datagramma IP in modalità transport. L'utilizzo di TCP nel payload IP è puramente esemplificativo. In giallo i componenti la cui integrità è protetta da AH.[2]

Contrariamente al caso precedente, nella modalità Tunnel viene protetto l'intero pacchetto IP di partenza, in quanto questo viene inglobato come payload di un nuovo pacchetto IP.
Per ulteriori dettagli si veda la figura successiva (i campi protetti da AH sono quelli evidenziati).

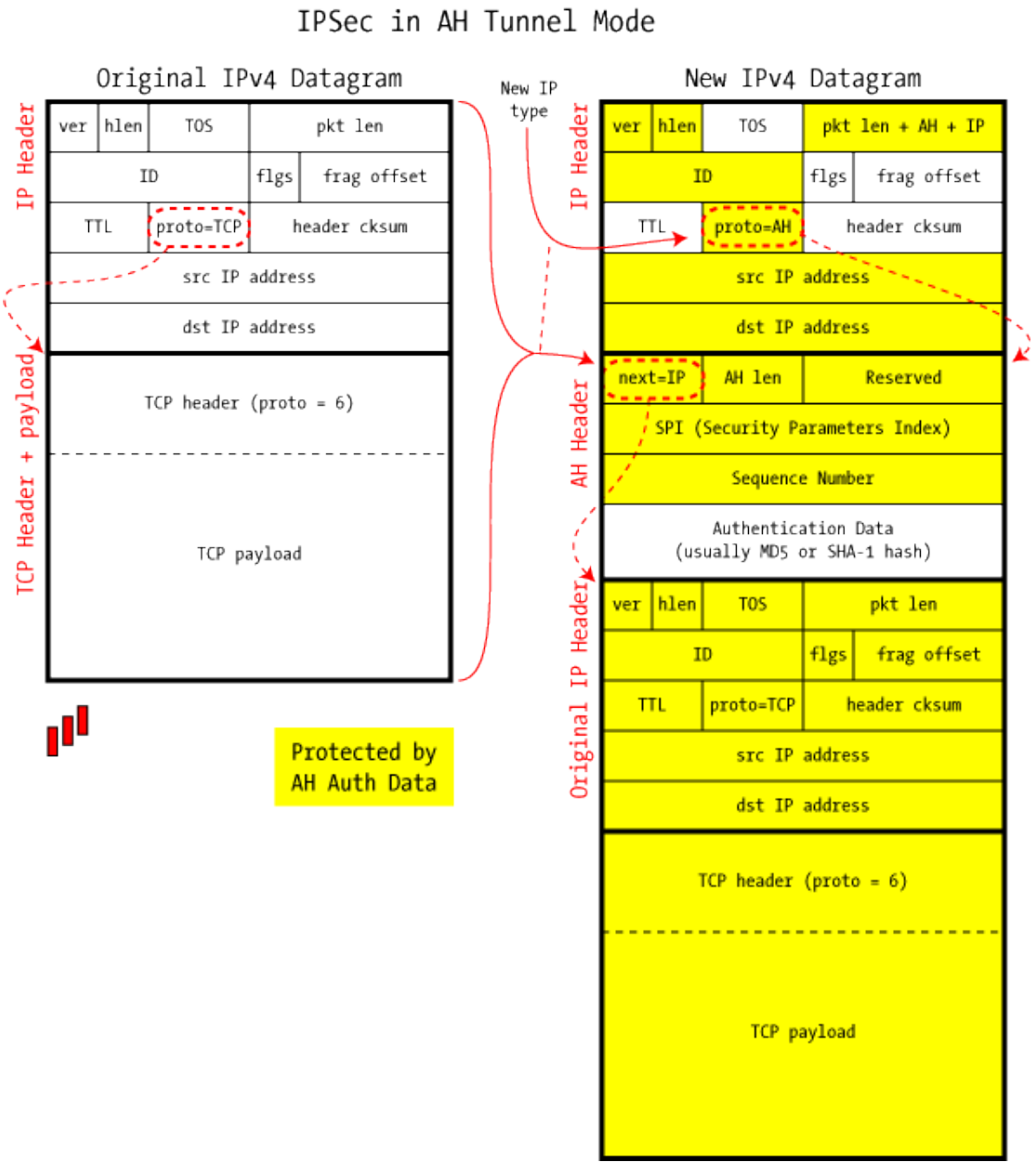


Figura 1.5: Inserimento di AH in un datagramma IP in modalità tunnel. L'utilizzo di TCP nel payload IP è puramente esemplificativo. In giallo i componenti la cui integrità è protetta da AH.[2]

1.4 Encapsulating Security Payload

Encapsulating Security Payload (ESP)[4] è un altro dei protocolli di IPsec, e fornisce i servizi di confidenzialità, autenticazione, controllo d'integrità e anti-replay. E' possibile configurare ESP in modo da avere confidenzialità senza autenticazione e controllo d'integrità.

1.4.1 Struttura

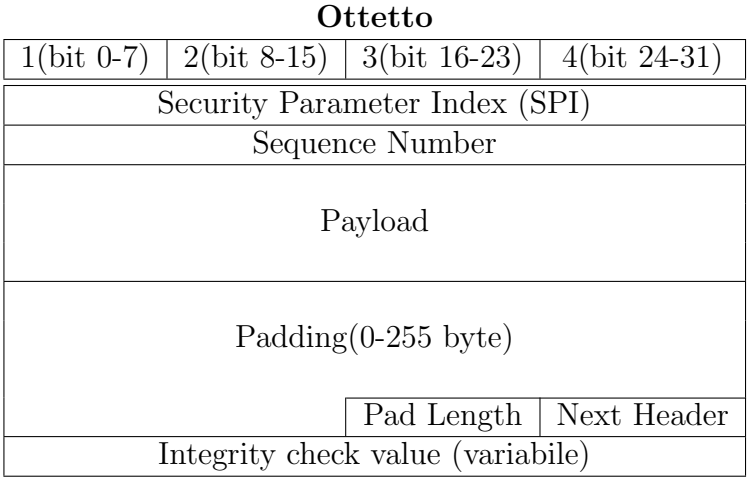


Figura 1.6: Encapsulating Security Payload

- **Security Parameter Index e Sequence Number:** hanno le stesse proprietà dei campi omonimi visti in precedenza.
- **Payload:** payload originario contenuto nel pacchetto IP, che include anche tutti i dati relativi alla sua protezione (es: Initialization vector[2]).
- **Padding:** padding opzionale da aggiungere al payload per due motivi principali:
  * alcuni algoritmi crittografici necessitano che il messaggio da crittare abbia una lunghezza pari a un multiplo di un certo valore;
  * la parte cifrata del pacchetto deve terminare con un word di 4 byte.
- **Next Header:** campo che identifica il payload successivo a ESP.
- **Pad Length:** campo che identifica la lunghezza del padding in byte.
- **Integrity check value (ICV):** valore a lunghezza variabile utilizzato per il controllo di integrità.

[2]Un valore, solitamente random, di lunghezza fissa necessario per alcuni algoritmi crittografici

1.4.2 ESP in tunnel e transport mode

Nella modalità Transport viene aggiunto l'header ESP prima del payload relativo al protocollo di livello superiore (es: TCP) e dopo l'header IP. In aggiunta, sono presenti in coda al payload il trailer ESP e i dati relativi all'autenticazione. Come si può notare nella figura sottostante, a differenza di AH, ESP non opera sull'header IP, lasciandolo inalterato.

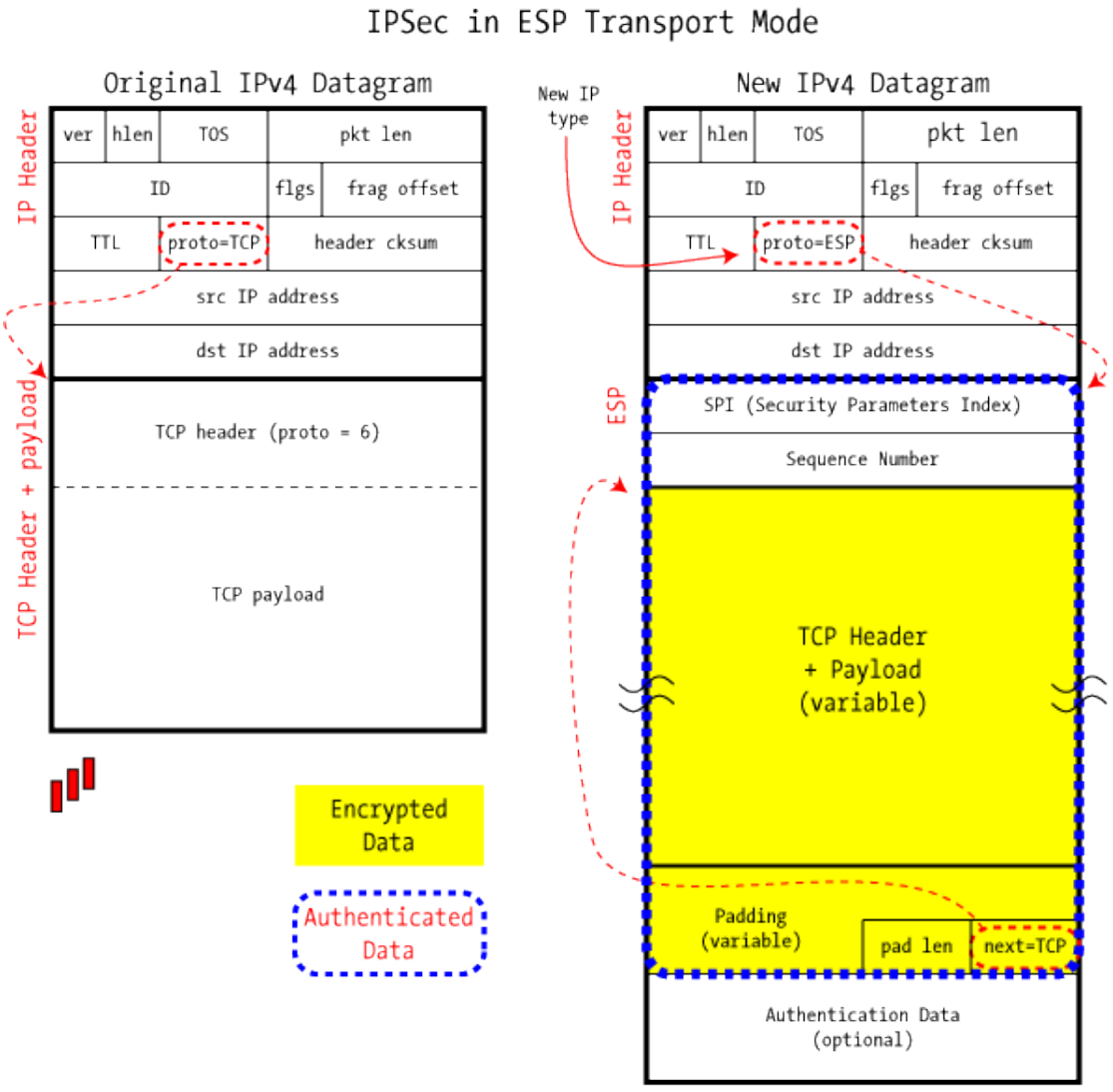


Figura 1.7: Inserimento di ESP in un datagramma IP in modalità transport. L'utilizzo di TCP nel payload IP è puramente esemplificativo. Il riquadro blu evidenzia i campi la cui integrità è protetta da ESP e in giallo i componenti crittati.[2]

Come per AH, nella modalità Tunnel viene protetto l'intero pacchetto IP di partenza, in quanto questo viene inserito come payload di un nuovo pacchetto IP.

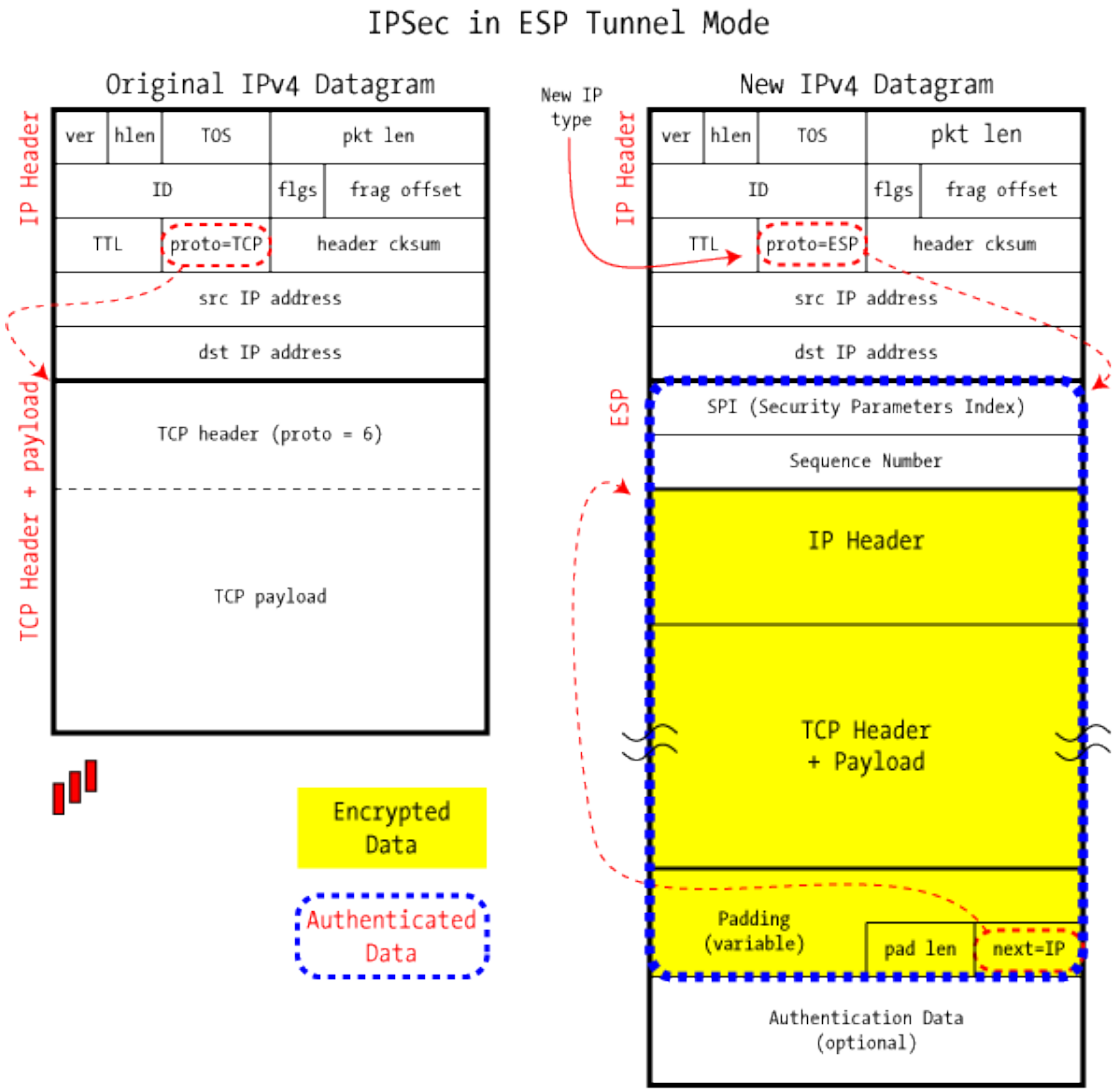


Figura 1.8: Inserimento di ESP in un datagramma IP in modalità tunnel. L'utilizzo di TCP nel payload IP è puramente esemplificativo. Il riquadro blu evidenzia i campi la cui integrità è protetta da ESP e in giallo i componenti crittati.[2]

1.5 Algoritmi di autenticazione

Per autenticare e verificare l'integrità di un pacchetto viene calcolato un Integrity check value (ICV) utilizzando un meccanismo chiamato Keyed-Hashed Message Authentication Code (HMAC). Esso utilizza una chiave segreta scambiata fra le due parti e una funzione di hash quale ad esempio SHA-1 o MD5.

MD5: MD5 (Message Digest algorithm 5) è un algoritmo crittografico di hashing realizzato da Ronald Rivest nel 1991. Questo tipo di codifica prende in input una stringa di lunghezza arbitraria e ne produce in output un'altra di 128 bit.

Tuttavia, oggi sono disponibili molti algoritmi per generare stringhe che collidono[3] e, pertanto, MD5 non è più considerato un algoritmo di hashing crittografico sicuro; nonostante questo MD5 è ancora largamente utilizzato.

SHA-1: SHA-1 (Secure Hash Algorithm 1) produce un output di 160 bit da un messaggio con una lunghezza massima di $2^{64} - 1$ bit ed è basato su principi simili a quelli usati da Ronald L. Rivest del MIT nel design dell'algoritmo MD5.
Come per MD5, anche SHA-1 non è risultato essere così sicuro come si pensava. Per questo motivo è stata creata un nuova versione chiamata SHA-2 ed è attualmente in sviluppo una terza versione SHA-3.

Calcolo dell'ICV. L'algoritmo per il calcolo dell'ICV è il seguente:

Algoritmo 1 HMAC(M, K)

Definiamo:
M: il messaggio da autenticare.
M′ : M diviso in blocchi da j bit.
opad: il byte `0x36` ripetuto $\frac{j}{8}$ volte.
ipad: il byte `0x5C` ripetuto $\frac{j}{8}$ volte.
K: la chiave di partenza.
h: la funzione di hash.
$\|$: l'operazione di concatenazione.

Algoritmo
if $length(K) > B$ **then** {le chiavi più lunghe di j bit vengono accorciate}
$K' = h(K)$
else if $K < B$ **then** {alle chiavi più corte di j bit viene aggiunto un padding}
$K' = K \| [$ `0x00` $* (B - length(K))]$
end if

$$HMAC_K(M) = h\Big((K' \oplus opad)||h\Big((K' \oplus ipad)||M'\Big)\Big)$$

1.6 Algoritmi di crittazione

Gli algoritmi crittografici vengono usati da IPsec in modalità ESP, per crittare i dati da inviare in modo che la comunicazione avvenga in maniera confidenziale.
I principali algoritmi crittografici utilizzati da IPsec sono DES, 3-DES e AES.

[3]Stringhe diverse che date come input generano lo stesso hash di output

DES. Il Data Encryption Standard (DES) è un algoritmo crittografico scelto come standard dal National Bureau of Standards (NIST) come Federal Information Processing Standard (FIPS) per il governo degli Stati Uniti d'America nel 1976 e in seguito diventato di utilizzo internazionale. Esso è basato su un algoritmo che utilizza una chiave simmetrica di 56 bit. Proprio per la scarsa lunghezza della chiave, DES è considerato attualmente poco sicuro dalla comunità internazionale.

3-DES. Triple Data Encryption Standard (3-DES) è un algoritmo crittografico derivato da DES. Esso consiste nell'applicare tre volte l'algoritmo DES su ogni blocco di dati e utilizza chiavi di 112 o 168 bit.

AES. L'Advanced Encryption Standard (AES) è un algoritmo crittografico attualmente utilizzato dal governo degli Stati Uniti. Utilizza chiavi lunghe 128,192 e 256 bit.

1.7 Internet Key Exchange v2

Internet Key Exchange v2 (IKEv2) è la seconda versione del protocollo IKE, la cui ultima revisione risale a Settembre 2010 [5].
IKEv2 viene utilizzata per instaurare le SA e per gestire le chiavi utilizzate dagli algoritmi per il controllo d'integrità, crittazione dei dati ecc. Per fare ciò, tutti i dati necessari (es: algoritmi crittografici, chiavi pubbliche) vengono negoziati e scambiati fra l'iniziatore della comunicazione e il ricevitore tramite alcuni scambi di messaggi. Tutti questi scambi sono composti da una richiesta e da una risposta. Verranno descritti ora i principali scambi compresi in IKEv2. Per indicare le SA utilizzate da IKE, verrà utilizzato il termine IKE SA, per indicare quelle utilizzate da AH ed ESP verrà utilizzato il termine CHILD SA. Inoltre le parti cifrate verranno incluse in un riquadro tratteggiato e i campi opzionali fra le parentesi quadre .

1.7.1 IKE SA INIT exchange

L'IKE SA INIT exchange è sempre il primo scambio di messaggi effettuato in una comunicazione utilizzante IKE.

Figura 1.9: IKE SA INIT exchange

- **HDR:** l'header IKE che include gli SPI scelti dall'initiatior e dal responder (pari a zero nel primo messaggio), il tipo di messaggio scambiato e altre informazioni come chi è l'initiator o il responder originale e se il messaggio è una richiesta o una risposta.
- **SAi1 e SAr1:** rispettivamente gli algoritmi crittografici supportati e/o preferiti dall'initiator e scelti dal responder sulla base di questi. Queste informazioni verrano poi utilizzate per la creazione della SA.
- **KEi e KEr:** contengono i valori necessari all'initiator e al responder per calcolare la chiave Diffie-Hellman.

- **Ni e Nr:** contengono i nonce dell'initiator e del responder. I nonce sono dei valori random utilizzati come input per gli algoritmi crittografici e per non permettere replay attack.

Dopo che l' IKE SA INIT exchange è andato a buon fine, viene creata la prima IKE SA ed entrambe le parti sono in grado di generare tutte le chiavi crittografiche necessarie per la protezione dell'IKE SA AUTH exchange e degli scambi successivi.

1.7.2 IKE SA AUTH exchange

L'IKE SA AUTH exchange contiene le informazioni necessarie ad autenticare le due parti e ad attuare un controllo d'integrità e avviene successivamente all'IKE SA INIT exchange.

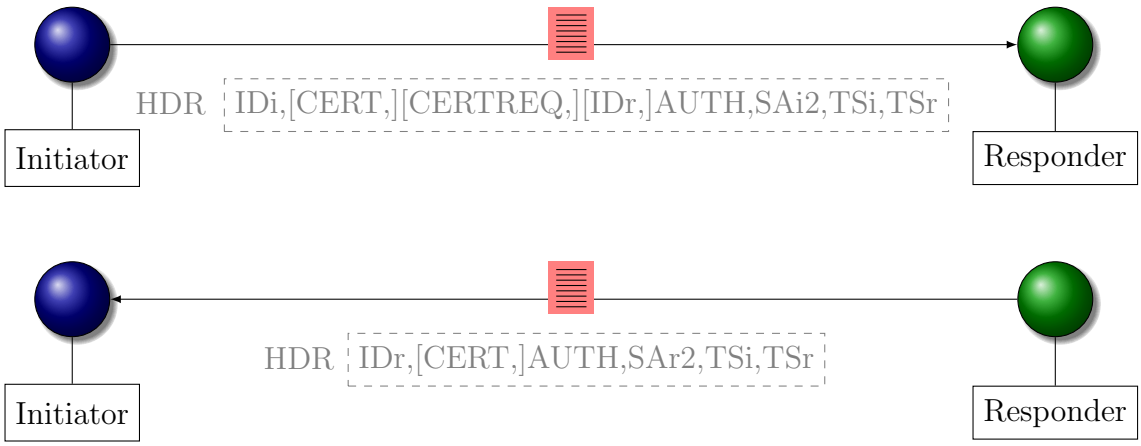


Figura 1.10: IKE AUTH exchange

- **HDR:** l'header IKE descritto in precedenza.
- **IDi e IDr:** contiene le informazioni necessarie a verificare l'identità dell'initiator e il responder.
- **CERT (opzionale):** contiene il certificato utilizzato per autenticare l'identità dell'initiator o responder.
- **CERTREQ (opzionale):** contiene le tipologie di certificati preferiti.
- **AUTH:** payload generato per il controllo d'integrità
- **SAi2 e SAr2:** rispettivamente gli algoritmi crittografici supportati e/o preferiti dall'initiator e scelti dal responder sulla base di questi. Queste informazioni verrano poi utilizzate per la creazione di una CHILD_SA.
- **TSi e TSr:** i traffic selector dell'initiator e del responder.

Se il metodo di autenticazione scelto non è EAP[4], le due parti vengono autenticate firmando un certo blocco di dati. Una volta completato lo scambio di messaggi, se tutto è andato a buon fine, viene creata la prima CHILD SA.

1.7.3 CHILD SA exchange

Se necessario è possibile la creazione di ulteriori CHILD SA tramite il CHILD SA exchange.

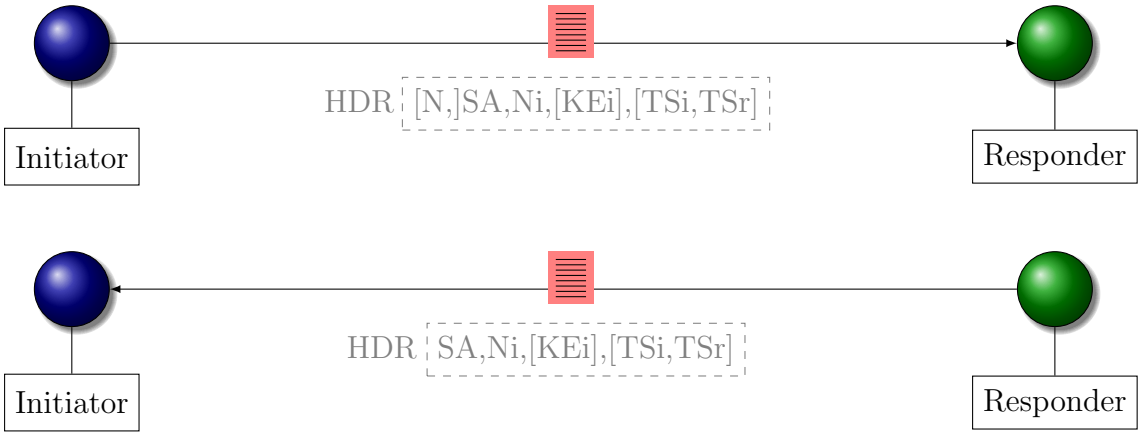


Figura 1.11: CHILD SA exchange

1.7.4 INFORMATIONAL exchange

L'INFORMATIONAL exchange può essere utile quando una delle due parti abbia necessità di controllare lo stato della comunicazione o di segnalare errori. Sono disponibili diversi payload per ogni situazione come ad esempio il NOTIFICATION payload (N), il CONFIGURATION payload (CP) e il DELETE payload (D). Per maggiori informazioni si veda la sez. 1.4 di [5]

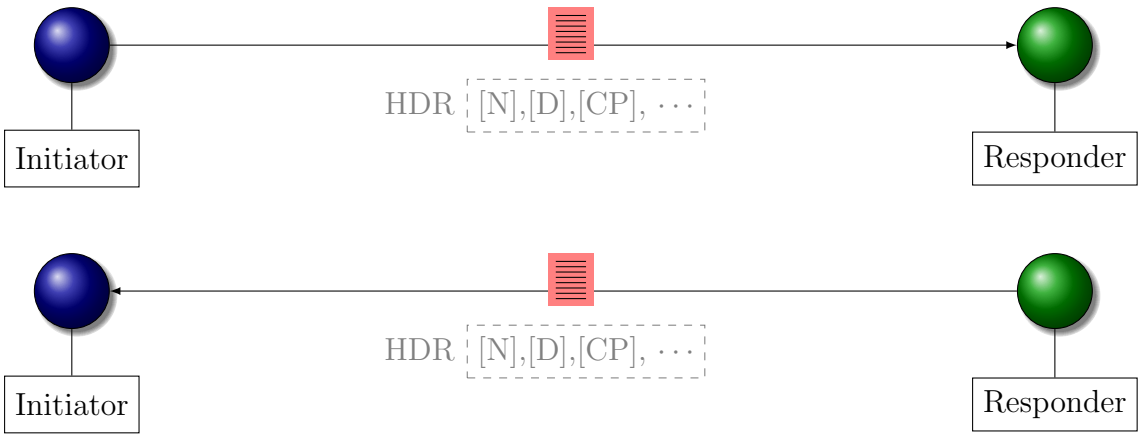


Figura 1.12: INFORMATIONAL exchange

[4]Extensible Authentication Protocol

1.7.5 Generazione delle chiavi

Uno dei compiti di IKE è di generare e gestire le chiavi necessarie per la protezione della comunicazione. La prima operazione che viene effettuata è calcolare la chiave Diffie-Hellman basandosi sui valori scambiati fra le due parti e contenuti nei payload KEi, KEr. Questo viene fatto sempre dopo L'IKE SA INIT exchange e può essere invece evitato nelle generazioni di chiavi successive.

Scambio di chiavi Diffie-Hellman

Diffie-Hellman è un protocollo crittografico che permette di generare una chiave condivisa e segreta su un canale pubblico senza che le due parti abbiano precondiviso alcun tipo di informazione. Il suo funzionamento è descritto nello schema seguente:

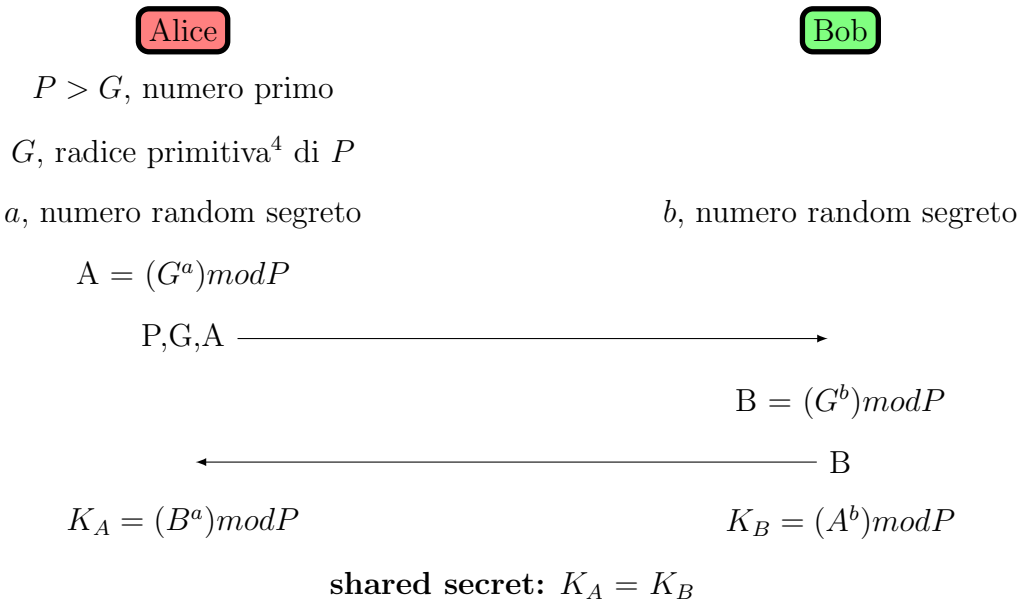


Figura 1.13: Scambio di chiavi Diffie-Hellman

La sicurezza di questo protocollo è di tipo computazionale, ovvero si basa sull'assunto che un attaccante con capacità computazionali limitate non sia in grado di calcolare o indovinare lo shared secret.
In questo caso il problema da risolvere è quello del logaritmo discreto: non conoscendo nè a nè b, in quanto non inviati singolarmente, un possibile malintenzionato dovrebbe ricavarli determinando il logaritmo discreto in base G di A e B, per il calcolo del quale non esiste ad oggi un algoritmo di complessità polinomiale. Inoltre, eccetto G, tutti gli altri valori utilizzati sono numeri molto grandi. Con dei normali computer diventa quindi molto difficile o addirittura impossibile

[4]Una radice primitiva mod(P) è un intero G le cui potenze (mod P) sono congruenti con i numeri coprimi (aventi massimo comune divisore pari a 1) a P.

risolvere questo problema in tempi ragionevoli. Considerando la velocità con cui cresce la capacità di calcolo delle CPU, questo potrebbe non essere più vero in futuro. In aggiunta, nulla esclude che qualcuno scopra un algoritmo polinomiale per questo problema.

Generazione chiavi per le IKE SA

Una volta calcolato lo shared secret Diffie-Hellman, non resta quindi che usarlo per generare tutte le chiavi. L'algoritmo è il seguente:

Algoritmo 2 Generazione chiavi per le IKE SA

Definiamo
$\|$: l'operazione di concatenazione.
SKEYSEED: il seed da cui verranno generate tutte le chiavi.
prf: la Pseudo-Random Function.
prf+: la Pseudo-Random Function plus, il cui output viene calcolato in questo modo:

prf+$(K, S) = T1\|T2\|T3\|\cdots$
Dove:
$T1$ =prf$(K, S\|$ `0x01`$)$
$T2$ =prf$(K, T1\|S\|$ `0x02`$)$
$T3$ =prf$(K, T2\|S\|$ `0x03`$)$

$\mathbf{g^{ir}}$: shared secret Diffie-Hellman.
Ni,Nr: nonce dell'initiator e del responder.
$\mathbf{g^{ir}}$: shared secret Diffie-Hellman.
$\mathbf{SK_{ei}}$ **e** $\mathbf{SK_{er}}$: chiavi usate per la crittazione.
$\mathbf{SK_{ai}}$ **e** $\mathbf{SK_{ar}}$: chiavi usate per il controllo d'integrità.
$\mathbf{SK_{pi}}$ **e** $\mathbf{SK_{pr}}$: chiavi usate per generare il payload AUTH.
$\mathbf{SK_{d}}$: valore usato per generare le chiavi per le CHILD SA.

Algoritmo
1 - SKEYSEED = prf$(Ni\|Nr, g^{ir})$

2 - keyBuffer = prf+(SKEYSEED, $Ni\|Nr\|SPIi\|SPIr$)

3 - $SK_d\|SK_{ai}\|SK_{ar}\|SK_{ei}\|SK_{er}\|SK_{pi}\|SK_{pr}$ = keyBuffer

Generazione chiavi per le CHILD SA

Per calcolare il buffer da cui ricavare le chiavi vengono svolte le seguenti operazioni. Per maggiori informazioni riguardo a come vengano estrapolate le chiavi dal buffer si veda la sez. 2.17 di [5].

Algoritmo 3 Generazione chiavi per le CHILD SA

Definiamo

SK_d : valore calcolato in precedenza, utilizzato per derivare le nuove chiavi per le CHILD SA.

prf+: la Pseudo-Random Function plus.

g^{ir}: lo shared secret Diffie-Hellman (solo se presente).

Ni,Nr: I nonce dell'initiator e del responder (scambiati durante l'IKE SA INIT).

KEYMAT: buffer contenente la concatenazione di tutte le chiavi.

$\|$: l'operazione di concatenazione.

$$\text{KEYMAT} = \text{prf} + (SK_d, [g^{ir}(nuovo)] \| Ni \| Nr)$$

Rigenerazione di chiavi

Ogni SA utilizzata da IKE, AH o ESP dovrebbe essere usata solamente per un certo ammontare di tempo. Questo limite rende necessario instaurare nuove SA e di conseguenza anche nuove chiavi. Il CHILD SA exchange può essere utilizzato per la generazione di nuove chiavi per le IKE SA e CHILD SA. Gli algoritmi sono gli stessi descritti in precedenza, con l'unica differenza che per le IKE SA il calcolo dello SKEYSEED viene svolto nel seguente modo:

$$\text{SKEYSEED} = \text{prf}(SK_d(vecchio), [g^{ir}(nuovo)] \| Ni \| Nr)$$

Nelle implementazioni minimali di IPsec, queste due possibilità possono non essere sviluppate. Se una SA termina il proprio tempo, verranno cancellate tutte le IKE SA e CHILD SA e ne verranno create di nuove.

1.8 Packet processing

1.8.1 Inbound processing

Quando un pacchetto viene ricevuto, questo viene elaborato da IPsec seguendo questi passi:

1. se il pacchetto contiene l'header IPsec, vengono letti i campi relativi all'SPI, alla destinazione del pacchetto e quale protocollo di sicurezza è stato usato per trovare una o più SA nell'SADB con queste caratteristiche. Si presentano quindi due casi distinti:
   (a) viene trovata una SA e vengono successivamente svolte le seguenti operazioni:
      i. se la protezione contro l'attacco anti-replay è attivata, viene controllato il valore del campo Sequence Number.
      ii. se attivato, viene effettuato il controllo d'integrità.
      iii. se il pacchetto è crittato, questo viene decrittato e ricostruito o estratto a seconda che si stia usando la modalità Transport o Tunnel.
      iv. il processo viene validato controllando che la SA sia stata usata correttamente, basandosi sulla policy relativa al pacchetto in entrata.
      v. il pacchetto viene passato al livello OSI superiore per ulteriori elaborazioni.
   (b) non viene trovata nessuna SA e il pacchetto viene scartato.
2. viene cercata una policy nell'SPDB per determinare quale delle tre azioni BYPASS, DISCARD e PROTECT è necessario operare sul pacchetto. Se l'azione prevista è PROTECT ma non è stata instaurata una SA, il pacchetto viene scartato.

1.8.2 Outbound Processing

Prima che un pacchetto venga inviato, questo viene processato basandosi sulla policy relativa. A questo consegue l'applicazione di una delle 3 azioni BYPASS, DISCARD o PROTECT. Il pacchetto viene quindi scartato, inviato senza utilizzare IPsec o protetto da IPsec.
Nell'ultimo caso verranno svolte altre operazioni:

1. se una SA è già stata creata, questa viene selezionata, altrimenti viene invocato IKE per creare una nuova SA.
2. la SA selezionata viene utilizzata per processare il pacchetto; se è richiesta la confidenzialità della trasmissione il pacchetto viene crittato. Inoltre, il valore HMAC, nel caso il controllo d'integrità sia attivato, viene calcolato e aggiunto al pacchetto

1.9 Critiche e debolezze

La principale critica che viene fatta ad IPsec riguarda la sua complessità. Ad esempio esistono due modalità, Tunnel e Transport, e due protocolli di sicurezza, AH e ESP che ci porta ad avere 4 possibili configurazioni: AH-Transport, AH-Tunnel, ESP-Transport e ESP-Tunnel. In aggiunta a questo, è possibile scegliere fra molte combinazioni di algoritmi crittografici e di autenticazione.
Inoltre ESP supporta anche la possibilità di fornire autenticazione e controllo d'integrità senza cifratura dei dati (usando il NULL encryption algorithm).
Tutte queste configurazioni possibili rendono molto complicato per un amministratore di rete comprendere quale sia la scelta migliore da effettuare nel suo caso. Alcuni esperti di sicurezza hanno riscontrato che alcune di queste configurazioni sono spesso insicure o addirittura inutili. Ad esempio, i ricercatori Niels Ferguson e Bruce Schneier hanno pubblicato nel 1999 una loro valutazione critica[6] del protocollo IPsec, in cui discutono alcune debolezze del protocollo e mettono in dubbio la reale utilità di alcune configurazioni.
Innanzitutto, viene messo in dubbio il fatto che sia realmente necessario avere due modalità, Transport e Tunnel, in quanto la prima risulta sostanzialmente un sottoinsieme della seconda e l'unico vantaggio che sembra avere è un overhead minore, ovviabile con una semplice compressione dell'header. Viene consigliato quindi di eliminare la modalità transport.
Viene inoltre discussa la differenza fra AH e ESP, e in particolare l'utilità del primo. AH sembra fornire un servizio di autenticazione migliore rispetto ad ESP, in quanto autentica sia il payload che l'header IP, però non fornisce nessun servizio di confidenzialità e, inoltre, in Tunnel Mode tutti i vantaggi vengono meno in quanto ESP critta e autentica l'intero pacchetto IP di partenza. In ogni caso, non viene fornita nessuna spiegazione nella documentazione di IPsec riguardo al perchè l'header IP debba essere autenticato. Venendo meno questa spiegazione e in base a quanto detto in precedenza riguardo alle modalità tunnel e transport, viene consigliato dagli autori di eliminare AH.
Nelle specifiche di ESP, è scritto che il servizio di autenticazione è opzionale. Questo rende quindi possibile una configurazione che permetta di ottenere una comunicazione cifrata ma senza nessuna autenticazione. Questo rende possibili alcuni attacchi come viene illustrato, ad esempio, in [7] e [8].
Un altra caratteristica di IPsec che viene criticata è l'unidirezionalità delle SA. Questo rende necessario la creazione di 4 SA differenti nel caso in cui si utilizzi AH insieme ad ESP. Considerando che la necessità di inviare dati in un unica direzione è abbastanza inusuale, Ferguson e Schneier propongono di utilizzare SA bidirezionali in modo da dimezzare il numero di SA necessarie.

Capitolo 2

Quantum Key Distribution

La Quantum Key Distribution (QKD), è una delle principali applicazioni della meccanica quantistica alle telecomunicazioni e all'informatica. Essa permette a due terminali la creazione di una chiave crittografica condivisa e segreta, fornendo una sicurezza di tipo incondizionato e garantendo la possibilità di rilevare un eventuale eavesdropper (Eve)[1].
In questo capitolo verrà descritto il suo funzionamento partendo dai principi fisici su cui si basa, e presentando successivamente un esempio di procotollo di QKD e alcune procedure di post-processing della chiave.

2.1 Principi Fisici

La meccanica quantistica è conosciuta per essere spesso contraria al modo di pensare comune, in quanto alcuni dei postulati su cui si basa possono risultare controintuitivi. Viste da un punto di vista quantistico, molte delle nostre certezze diventano invece incertezze e azioni che consideriamo possibili, impossibili.
Alcuni di questi postulati, che ad una prima analisi possono apparire delle limitazioni, sono risultati invece utili nel tempo a sviluppare dei sistemi di sicurezza che si basino proprio su queste incertezze e/o impossibilità.

Principio di indeterminazione di Heisenberg

Supponendo di voler misurare la posizione e la quantità di moto di un oggetto, siamo soliti pensare che la precisione della nostra misurazione dipenda solamente dagli strumenti in nostro possesso. In altre parole, più gli strumenti sono precisi, più la nostra misura è vicina al valore reale.
In generale, possiamo affermare che qualunque coppia di grandezze osservabili generiche, che non siano nella relazione di essere compatibili, non si potranno misurare simultaneamente, se non a prezzo di un'indeterminazione sull'una che

[1] Letteralmente dall'inglese "ficcanaso". Rappresenta una terza entità che si frappone fra le due parti della comunicazione cercando di leggere o modificare i dati trasmessi senza il loro permesso.

è tanto più grande quanto più piccola è l'indeterminazione sull'altra grandezza e l'atto stesso di misurarle porta una perturbazione nel sistema.
Questa affermazione porta il nome di "Principio di indeterminazione di Heisenberg" ed è stata dimostrata[2] nel 1927 da Werner Karl Heisenberg.

Teorema di No-cloning quantistico

Il teorema di no-cloning quantistico è un altro dei postulati della meccanica quantistica e afferma che *non è possibile duplicare esattamente (cloning) uno stato quantistico*[3] *sconosciuto a priori.*
Questo teorema è stato dimostrato da William Wotters e Wojciech Zurech nel 1982[4]. Per comprendere il significato di "duplicare" in questo caso, si immagini di avere una macchina che prenda come input uno stato quantistico e ne restituisca due copie identiche.

2.2 Protocollo BB84

2.2.1 Descrizione

Il primo protocollo di QKD è stato proposto nel 1984 da Charles H. Bennet e Gilles Brassard con il nome BB84[9]. Il suo funzionamento è descritto qui di seguito.
Chiamiamo Alice e Bob le due entità che vogliono creare una chiave crittografica segreta e condivisa. Fra di esse sono presenti due canali: uno pubblico ordinario (es: Cavo ethernet) e uno "quantistico"(es: fibre ottiche o spazio libero) in cui sia possibile inviare fotoni. Potendo considerare i fotoni come oggetti "quantistici", essi sono soggetti alle leggi fisiche sopra descritte. Consideriamo ora 4 stati quantistici definiti da fotoni polarizzati a 0°,90°,45°e 135°e rappresentiamoli rispettivamente con i simboli $\uparrow$, $\rightarrow$, $\nearrow$ e $\searrow$. Gli stati $\uparrow$ e $\rightarrow$ identificano una base che indicheremo con $+$; a loro volta $\nearrow$ e $\searrow$ ne indentificano un'altra che indicheremo con $\times$.
Successivamente, Alice sceglie una stringa di bit e una sequenza di basi casuali con cui codificarla e invia a Bob la corrispondente sequenza di fotoni polarizzati. Per ciascuna base, Alice associa i due stati di polarizzazione al bit 0 o al bit 1, rispettivamente. La codifica di un bit mediante un fotone polarizzato rappresenta un "qubit" (quantum bit). Un esempio di schema può essere il seguente:

[2] W. Heisenberg, (1927), *"Über den anschaulichen Inhalt der quantentheoretischen Kinematik und Mechanik"*, *Zeitschrift für Physik* 43 (3–4): 172–198

[3] Maggiori informazioni su G. Auletta, M. Fortunato e G.Parisi, *"Quantum Mechanics"*, Cambridge University Press, 2009.

[4] W. K. Wootters and W.H. Zurek, *"A Single Quantum Cannot be Cloned"*, *Nature* 299 (1982), pp. 802–803

Base	0	1
+	→	↑
×	↗	↘

Figura 2.1: Basi e rappresentazione dei bit scelti da Alice.

Per ciascun fotone ricevuto, Bob, sceglie casualmente e indipendentemente da Alice una base con cui misurarne lo stato e, concordemente, associa il risultato al bit 0 o al bit 1, rispettivamente. E' importante notare che, come conseguenza del principio di indeterminazione di Heisenberg, nel caso Bob scelga di misurare un fotone polarizzato secondo la base + utilizzando la base ×, il risultato sarà casuale, e la probabilità che sia corretto è del 50%. Inoltre l'informazione ricevuta da Bob viene ulteriormente penalizzata dal fatto che alcuni fotoni possono non arrivare a destinazione.

I passi successivi del protocollo hanno luogo sul canale pubblico, in cui Bob invia ad Alice la sequenza delle basi utilizzate per misurare i fotoni ricevuti, in modo da verificare tramite una risposta di Alice, quali di queste sono le stesse da lei utilizzate. La chiave (raw key) sarà quindi composta dai bit correttamente misurati. Nella figura sottostante viene riportata una possibile sessione del protocollo BB84.

Comunicazione Quantistica								
Bit casuali di Alice	0	1	1	0	1	0	0	1
Basi casuali di Alice	+	+	×	+	×	×	×	+
Fotoni polarizzati inviati da Alice	↑	→	↘	↑	↘	↗	↗	→
Basi casuali di Bob	+	×	×	×	+	×	+	+
Bit ricevuti da Bob	0	0	1			0	1	1
Discussione pubblica								
Bob segnala ad Alice le basi utilizzate	+	×	×			×	+	+
Alice segnala a Bob quali erano corrette	ok		ok			ok		ok
Informazione condivisa	0		1			0		1
Raw Key	0		1			0		1

Figura 2.2: Funzionamento del protocollo BB84

2.2.2 Considerazioni sull'Eavesdropping

Uno dei punti di forza del protocollo BB84, o più in generale, di tutti i protocolli utilizzanti la trasmissione di stati quantistici, è l'impossibilità per Eve di "ascoltare" la comunicazione, ovvero di misurare lo stato del fotone e rinviarlo a Bob, senza perturbare statisticamente il sistema e permettere perciò ad Alice e Bob di accorgersi dell'intruso. Alice e Bob possono infatti rivelare alcuni dei bit delle loro stringhe finali, e se questi corrispondono, dedurre che con una buona probabilità nessuno si è frapposto nella loro comunicazione. Infatti seguendo

quanto esposto in precedenza, considerando i risultati derivanti dalla meccanica quantistica, possiamo seguire la sequente logica,

$$\text{Nessuna perturbazione} \Longrightarrow \text{Nessuna misurazione} \Longrightarrow \text{Nessun eavesdropper},$$

e affermare che se non sono presenti errori, significa che non siamo in presenza di nessun eavesdropper.
Questo risulta vero a meno che Eve non applichi questa strategia solamente ad una piccola frazione dei bit trasmessi. In tal caso, l'errore prodotto risulterebbe molto limitato, e diventerebbe quindi arduo per Alice e Bob rilevare la presenza di un eavesdropper.
Un altro vantaggio è dato dal fatto che per il teorema di no-cloning quantistico, Eve non è in grado di copiare perfettamente il fotone inviato da Alice (non conoscendolo a priori) e misurarlo separatamente una volta che Bob ha annunciato le basi usate.

2.3 Operazioni di post-processing

Poichè, nel canale quantistico, includendo i dispositivi di trasmissione e ricezione, è sempre presente un certo grado di distorsione e di rumore, anche in assenza di eavesdropper, i bit ricevuti da Bob possono presentare degli errori.
Un metodo per risolvere questi problemi consiste nell'applicare due processi chiamati Information Reconciliation e Privacy Amplification, entrambi descritti in [10].
Essi fanno uso di una comunicazione bidirezionale (discussione) tra Alice e Bob, attraverso un canale pubblico.

2.3.1 Information Reconciliation

L'Information Reconciliation è la procedura che permette ad Alice e Bob di correggere le possibili differenze fra i bit inviati da Alice e i corrispondenti bit ricevuti da Bob. Supponendo che Eve sia in grado di ascoltare tutte le comunicazioni pubbliche, Alice e Bob devono riconciliare le proprie chiavi senza rivelare troppa informazione. Ci sono vari algoritmi per questo scopo; qui si descrive il protocollo "Cascade".
Essi scelgono una permutazione casuale di posizioni di bit (per rendere casuale la localizzazione degli errori) e dividono la stringhe risultanti in blocchi di b bit, dove b è scelto in modo tale da rendere il più possibile improbabile la presenza di più di un errore per blocco. A questo punto Alice e Bob effettuano un controllo di parità[5] per ogni blocco di bit confrontandosi ogni volta. Nel caso vengano riscontrati errori su un blocco, viene ripetuta iterativamente l'operazione bisecandolo in blocchi più piccoli e effettuando il controllo su di essi fino a trovare l'errore. Per non permettere ad Eve di essere in grado di dedurre qualcosa da questo processo,

[5]Procedimento utilizzato nei calcolatori per rilevare errori nella trasmissione o nella memorizzazione dei dati

viene tolto il bit finale di ogni blocco nel quale è stato trovato l'errore.
Qualsiasi sia il valore di b, è altamente probabile che rimangano ulteriori discrepanze fra i bit che compongono la stringa di Alice e quella di Bob, in conseguenza del fatto che tale protocollo è in grado di correggere un errore per blocco.
Per correggere questi errori addizionali, viene ripetuto il processo descritto in precedenza aumentando progressivamente il valore b, fino a quando la probabilità d'errore è sufficientemente bassa. Ora, continuare ad usare sempre questa strategia risulterebbe inefficiente perchè forzerebbe Alice e Bob a eliminare almeno un bit per ogni blocco per garantire la privacy della comunicazione. Supponendo ad esempio di avere lasciato esattamente 2 errori e di avere t blocchi, la probabilità che rimangano errori è $\frac{1}{t}$ con un costo di t bit. E' necessario quindi, pensare ad una nuova strategia che diminuisca la probabilità di non rilevare gli errori rimanenti. Si potrebbe infatti considerare che Alice e Bob scelgano un sottoinsieme di posizioni random nelle loro stringhe e che effettuino un controllo di parità sulle sottostringhe formate dai bit presenti in quelle posizioni. Se le stringhe di partenza non sono identiche la probabilità che siano rilevati errori nelle sottostringhe è pari ad $\frac{1}{2}$. Se questi vengono rilevati, i passi successivi sono simili a quelli della strategia precedente. Viene effettuato infatti iterativamente un controllo di parità bisecando la stringa e eliminando l'ultimo bit, con l'unica differenza che ogni controllo viene svolto su sottostringhe formate con il procedimento appena descritto (ogni sottoinsieme random viene scelto sulle stringhe risultanti dal processo di bisezione). Con questo sistema la probabilità che rimangano errori è pari a 2^{-t}, sempre ad un costo di t bit.
Una volta che l'ultimo errore viene rilevato, viene ripetuta l'ultima procedura per un certo numero di iterazioni per assicurare che le stringhe possedute da Alice e Bob sono di fatto identiche con una probabilità trascurabile che ulteriori errori non siano stati rilevati.

2.3.2 Privacy Amplification

Dopo aver terminato con successo la fase di Information Reconciliation, Alice e Bob condividono con alta probabilità due stringhe (chiavi) identiche, ma su cui l'avversario ha una quantità di informazione non trascurabile. Ad esempio la presenza di Eve potrebbe non essere stata rilevata in quanto gli errori da lei indotti potrebbero essere stati "coperti" da quelli dovuti al rumore presente sul canale o a imperfezioni della strumentazione utilizzata.
Inoltre, anche se durante la fase di Information Reconciliation Eve non recuperasse nessuna informazione riguardo alla stringa inviata da Alice, in quanto l'ultimo bit di ogni sottostringa viene sempre scartato ad ogni controllo di parità, essa potrebbe convertire l'informazione sui bit fisici della stringa in informazione sui bit della chiave finale. Per esempio se Eve conoscesse il bit a della stringa S e Alice e Bob rivelassero il bit di parità di S scartando il bit a, essa conoscerebbe la parità dei bit rimanenti. Se consideriamo che Eve è in grado di ricavare un bit di parità di una stringa quando conosce la parità di un sottoinsieme non vuoto di quella stringa, deduciamo che se Eve è in grado di conoscere al massimo k bit

fisici della chiave, essa conoscerà al massimo k bit della stringa dopo la fase di reconciliation.
A questo punto Alice e Bob sono in grado di svolgere l'operazione chiamata "Privacy Amplification". Denominiamo S_r la stringa ottenuta dopo l'Information Reconciliation e n la sua lunghezza. Se Eve conosce al massimo k bit deterministici[6] della stringa S_r, viene scelta pubblicamente una funzione di hash casuale $h(S_r)$ da una classe di funzioni definite di Universal Hashing[7],

$$h(S_r) : \{0,1\}^n \rightarrow \{0,1\}^{n-k-s},$$

in modo che, scegliendo un valore s opportuno, si diminuisca l'informazione media mutua ricavabile da Eve.

[6]Un bit deterministico di informazione riguardo ad una stringa S è il valore di output di una generica funzione $e(x) : \{0,1\}^n \rightarrow \{0,1\}$. Vedi [10]

[7]J. Lawrence Carter, Mark N. Wegman, *"Universal classes of hash functions"*, *Journal of Computer and System Sciences*, Volume 18, Issue 2, Aprile 1979, Pagine 143-154

Capitolo 3

Progettazione

Gli algoritmi per lo scambio di chiavi crittografiche condivise, necessitano solo di normali computer e di un canale ordinario in quanto si basano principalmente su dei calcoli matematici da svolgere. Si veda ad esempio quanto detto in precedenza nel capitolo 1 riguardo al protocollo Diffie-Hellman.
Volendo integrare un protocollo di QKD con un qualsiasi protocollo di sicurezza, e utilizzarlo in sostituzione o in aggiunta a questi algoritmi, il primo problema che si riscontra è il bisogno di un ulteriore canale, quello quantistico, realizzabile con un canale a fibre ottiche o in spazio libero. Oltre a questo, è necessaria una strumentazione ulteriore per gestire l'invio, la ricezione e l'elaborazione dei fotoni, agente in maniera completamente indipendente e con tempistiche spesso più lente dei computer su cui opera il protocollo di sicurezza, nel nostro caso IPsec.
La valutazione e progettazione di questa integrazione, dovrà tenere conto quindi di questi problemi e sviluppare un sistema che garantisca una buona sincronizzazione fra IPsec e sistema QKD, senza perdere le migliorie che questa innovazione comporta. Nel nostro caso si è partiti da un'idea di base scelta nella letteratura recente e la si è successivamente analizzata e sviluppata.

3.1 L'idea di S. Nagayama e R. Van Meter

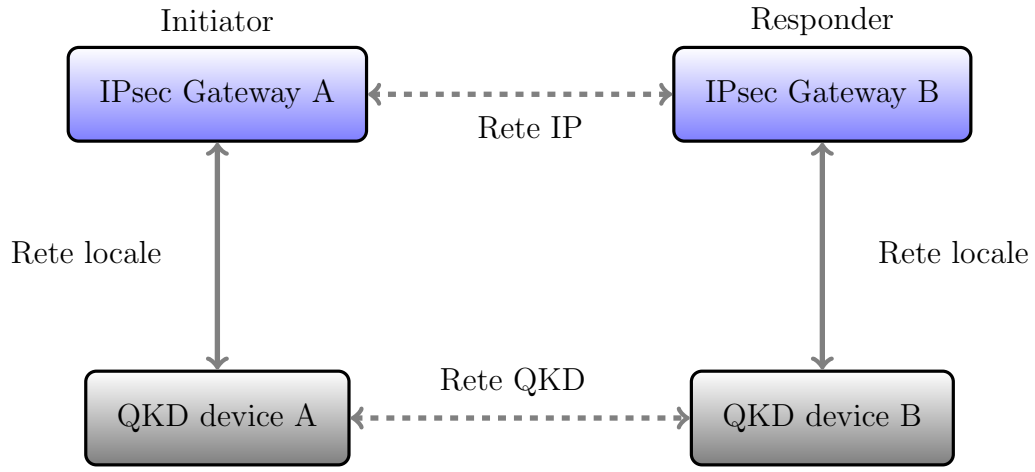
Nell'Ottobre del 1999, i ricercatori Shota Nagayama e Rodney Van Meter hanno pubblicato un Internet-Draft[1][11] che propone una possibile modalità di integrazione fra la QKD e IPsec, e più in particolare con IKE.

3.1.1 Architettura

L'architettura del sistema proposta dai due ricercatori, è descritta nella figura sottostante. Ogni parte è provvista di un Gateway IPsec e da un sistema QKD. I Gateway sono collegati tramite una rete IP e i sistemi QKD mediante una rete

[1]Una serie di documenti pubblicati da IETF. Solitamente sono delle bozze di Request for comments (RFC).

QKD. Infine i rispettivi sistemi QKD e Gateway IPsec sono collegati tramite una rete "locale", assumendo che quest'ultima sia sicura.



3.1.2 Modifiche ad IKE

Il documento prevede alcune modifiche e aggiunte al protocollo IKE (v2). In particolare è necessario lo scambio di due tipi di informazione: un Key ID e dei metodi di Fallback. Il primo rappresenta l'identificativo di un blocco di bit, condivisi e prodotti tramite la rete QKD, abbastanza grande da essere utilizzata come chiave crittografica. Il secondo rappresenta invece dei metodi di "ripiego" per gestire l'eventualità che il sistema QKD non sia in grado di fornire una nuova chiave.
Entrambe le informazioni sono contenute in due nuovi payload che verranno ora descritti. Se non diversamente specificato, il funzionamento e le varie notazioni di IKE restano le medesime viste nel primo capitolo.

Payload QKD Key ID

La struttura del payload è descritta nella figura seguente.

0 7	8 15	16 31	
Reserved	C	Reserved	Payload length
version	N	flags	reserved
Key ID			

Figura 3.1: Payload QKD Key ID

I primi 32 bit identificano l'header generico di IKE. Il critical bit (C) viene posto ad uno per evitare replay attack.

- **version:** indica la versione del protocollo.

- **N:** rappresenta il No Key bit. Viene posto a 1 quando non siamo in grado di reperire la chiave da nostro sistema QKD. Questo campo deve essere pari a 0 durante l'IKE SA INIT.

- **flags:** contiene dei flag bit. Questo campo è posto pari a 0.

- **Key ID:** rappresenta l'ID della chiave che vogliamo utilizzare.

Payload QKD Fallback

Il payload QKD Fallback è descritto dalla figura seguente.

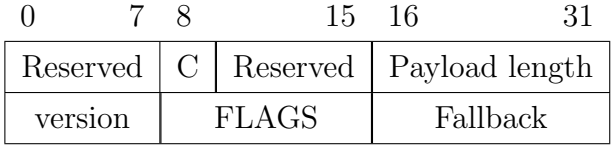


Figura 3.2: Payload QKD Fallback

I primi 32 bit e i campi version e FLAGS hanno le stesse proprietà del payload QKD Key ID. Il campo fallback contiene le informazioni riguardo al metodo di fallback scelto. Nel documento sono descritti 3 possibili metodi di fallback, elencati qui di seguito:

1. **WAIT_QKD:** indica che IKE deve aspettare che il sistema QKD gli fornisca una nuova chiave, a meno che non sia scaduto il lifetime della chiave. In questo caso la comunicazione viene interrotta.

2. **DIFFIE-HELLMAN:** indica che IKE deve utilizzare Diffie-Hellman per la generazione delle chiavi come descritto nel primo capitolo.

3. **CONTINUE:** indica che IKE può continuare ad utilizzare la chiave più recentemente utilizzata.

Modifiche all'IKE SA INIT exchange

L'IKE SA INIT exchange funziona sostanzialmente come descritto nel primo capitolo con l'unica differenza che i payload KEi/r e Ni/r non vengono inviati in quanto vengono utilizzati solamente con il protocollo Diffie-Hellman e non dalla QKD. Inoltre l'initiator deve necessariamente aspettare che il sistema QKD gli fornisca una chiave e non ha la possibilità, quindi, di utilizzare metodi di Fallback.
Il nuovo IKE SA INIT exchange è mostrato nella figura successiva. Si noti che il payload QKD key ID è sempre lo stesso in quanto il responder deve essere necessariamente d'accordo con l'initiator su quale Key ID impiegare.

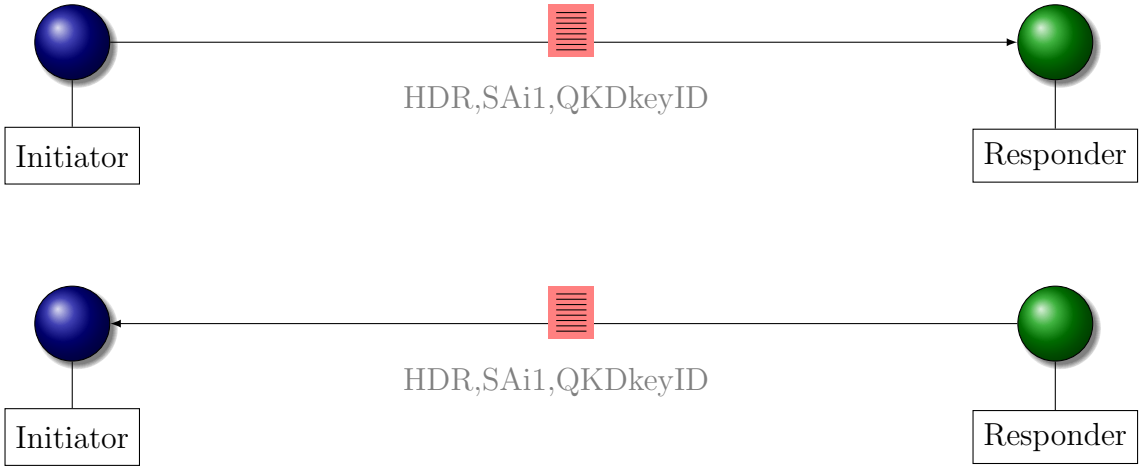


Figura 3.3: IKE SA INIT exchange con QKD

Modifiche all'IKE AUTH exchange

Come per il caso precedente, anche qui non ci sono modifiche particolari se non l'aggiunta del payload QKD fallback (crittato), che contiene il metodo di Fallback proposto dall'initiator. Se il responder non è d'accordo nell'usare questo metodo di Fallback deve rispondere con un errore.

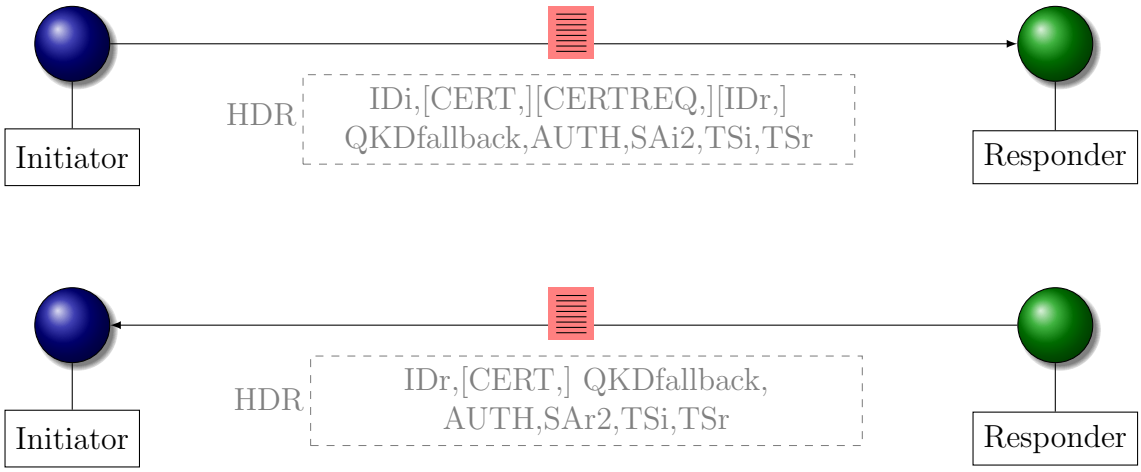


Figura 3.4: IKE AUTH exchange

Rigenerazione delle chiavi

Per rigenerare nuove chiavi (rekeying) con la QKD è opportuno utilizzare il CREATE CHILD SA exchange piuttosto che ripetere gli scambi iniziali IKE SA INIT e IKE SA AUTH, in quanto risulta molto più semplice operare in questo modo.
Il nuovo CREATE CHILD SA exchange è mostrato in figura.

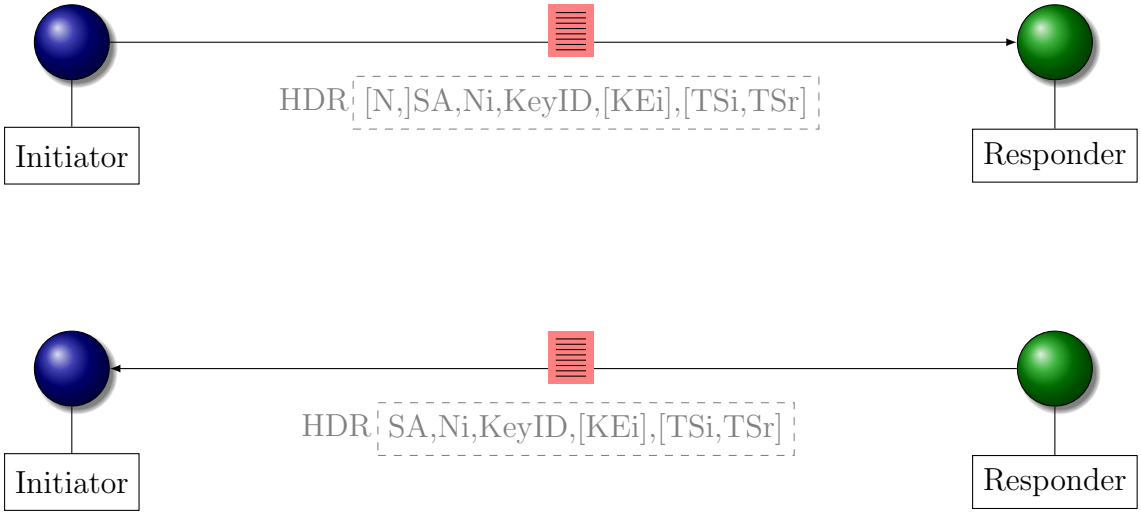


Figura 3.5: CREATE CHILD SA exchange con QKD

Il sistema dovrebbe cercare di usare sempre la QKD per l'ottenimento delle chiavi. Quando questo è possibile, il payload KeyID va a contenere l'ID della nuova chiave da utilizzare, il suo NoKey bit viene posto a 0 e i payload KEi e KEr vengono omessi.
Nel caso non sia invece possibile reperire nuove chiavi tramite QKD, si deve utilizzare un metodo di Fallback, se deciso in precedenza. L'initiator deve inviare il QKD Fallback payload con il No Key bit posto a 1 e il QKD KeyID payload con i campi No Key e KeyID posti rispettiamente a 1 e 0. Il responder deve rispondere con gli stessi payload. Se il metodo di fallback scelto è Diffie-Hellman, vengono inviati anche i payload KEi e KEr.
Si noti che mentre nell'IKE SA INIT exchange, l'uso dei nonce non è necessario con la QKD, durante il CREATE CHILD SA exchange risulta importante inviarli per prevenire eventuali replay attack.

Considerazioni

Il documento di Nagayama e Van Meter, essendo solamente una bozza (draft) e non un RFC[2] compiuto, fornisce solamente un'idea di base da cui partire e un approccio con cui affrontare il problema dell'integrazione fra IPsec e QKD.
E' stato necessario quindi, in fase di progettazione, definire, a partire da questa idea, delle specifiche il più possibile complete e precise su cui basare la nostra implementazione, a volte modificando l'idea di base dei due ricercatori. Verranno descritte ora tutte le considerazioni, modifiche e definizioni derivanti da questo lavoro.

[2]Request for Comments.

3.2 Sviluppo dell'architettura del sistema

3.2.1 KeyID e Key Unit

Come KeyID abbiamo scelto di usare un valore intero senza segno diverso da 0, in quanto, come descritto in precedenza, quest'ultimo valore viene utilizzato quando le chiavi QKD non sono reperibili e non può quindi essere compreso in altri scopi. Ad ogni KeyID è associato una Key Unit, che rappresenta una singola stringa di bit che abbia una lunghezza il più possibile vicina al massimo comune divisore delle lunghezze dei valori di input degli algoritmi utilizzati da IPsec. In questo modo, il sistema è in grado di adattarsi a seconda dell'insieme di algoritmi utilizzati dalle due parti, scegliendo semplicemente una chiave composta da N Key Unit, dove

$$N = \left\lceil \frac{L_{alg}}{L_{keyUnit}} \right\rceil$$

e L_{alg} e $L_{keyUnit}$ sono rispettivamente la somma delle lunghezze dei valori di input degli algoritmi utilizzati da IPsec e la lunghezza scelta per i Key Unit. Ad esempio nel caso volessimo usare AES-256 e Key Unit da 64bit, il numero di Key Unit necessarie risulta essere $\left\lceil \frac{256}{64} \right\rceil = 4$. Considerando gli algoritmi previsti da IPsec, abbiamo scelto di utilizzare proprio Key Unit da 64bit come nell'esempio.

3.2.2 QKD device

Nel documento non viene fornita alcuna informazione riguardo a come debba essere gestita la comunicazione fra QKD device e Gateway IPsec, e a come debbano essere organizzate le Key Unit e i rispettivi KeyID.
E' stato quindi definito il QKD device come l'insieme di 3 entità: un QKD server, un database e un sistema QKD. Ognuna di queste entità è collegata alle altre come mostrato in figura.

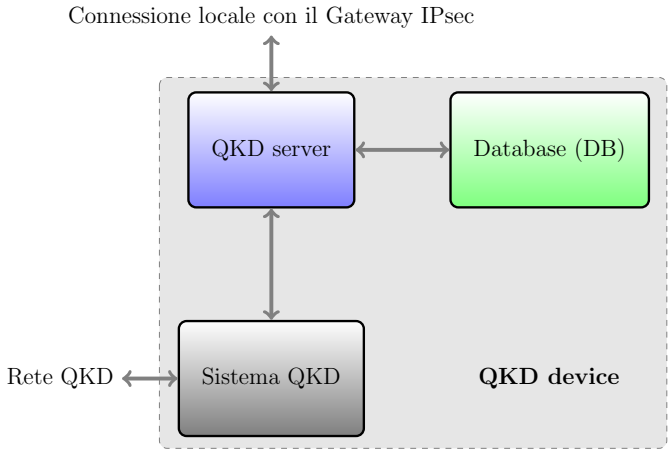


Figura 3.6: QKD device

Il ruolo svolto da ogni componente è il seguente:

- **QKD server:** E' un server generico che ha il compito di gestire la comunicazione con l'esterno e quindi di gestire le richieste da parte del Gateway IPsec. Inoltre si occupa di prelevare le chiavi grezze prodotte dal sistema QKD e di scrivere/leggere queste sul database.
  Ad ogni chiave viene associato un Key ID secondo l'ordine con cui vengono prodotte e successivamente scritte sul database. Ad esempio la prima chiave inserita avrà Key ID uguale a 1, la seconda uguale a 2 e così via. Una volta che viene raggiunto il valore massimo pari a 2^{32} o ad un valore minore che può essere scelto a priori, il conteggio del Key ID riparte da 1. Inoltre, quando questo accade, le nuove chiavi vengono inserite sostituendo le chiavi giù utilizzate, a partire dalla chiave utilizzata con Key ID più piccolo.

- **Database (DB):** Memorizza logicamente le chiavi prodotte dal sistema QKD. Ogni entry al suo interno è provvista dei parametri:

Key ID	Key Unit Value	Flag	Timestamp

Figura 3.7: Entry Database QKD

  dove:

  * Key ID = l'ID della chiave descritto in precedenza.
  * Key Value = il valore del Key Unit.
  * Flag = un bit flag per indicare se la chiave è già stata usata in precedenza.
  * Timestamp = l'istante in cui la chiave è stata scritta sul database.

- **Sistema QKD:** realizza il sistema QKD secondo quanto descritto nel secondo capitolo.

Potendo scrivere o leggere sul database in qualsiasi momento (a meno che non abbia raggiunto la capienza massima), risulta evidente che uno dei principali vantaggi di questa struttura è la possibilità di far funzionare IKE con chiavi QKD anche se il sistema QKD risulta "spento" oppure di salvare le chiavi prodotte dal sistema e dalla rete QKD anche se non immediatamente utilizzate.

Il QKD server deve essere in grado di disattivare o attivare il sistema QKD, in modo che non continui a produrre chiavi anche se queste non possono essere scritte sul database.

3.2.3 Protocollo fra QKD device e Gateway IPsec

Per gestire la comunicazione fra QKD device e Gateway IPsec, è stato definito un semplice protocollo per fornire una specifica su quali dati è necessario inviare e ricevere e in che modo inviarli.

La trasmissione dei dati avviene tramite protocollo TCP che garantisce maggior affidabilità rispetto ad altri protocolli quali ad esempio UDP. I payload principali sono i seguenti:

* **QKD KEY REQUEST**

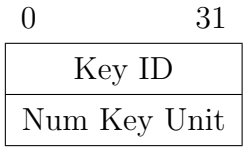


Figura 3.8: QKD KEY REQUEST

Viene inviato dal Gateway IPsec al QKD device per richiedere nuovo materiale da utilizzare come chiave crittografica. Il campo Key ID rappresenta il primo Key ID da cui partire per il reperimento delle Key Unit. In altre parole se ad esempio viene inviato questo payload con i valori Key ID e Num Key Unit pari a 1 e 5, verranno inviate le Key Unit con Key ID da 1 a 5.
Nel caso in cui il Gateway non voglia specificare alcun Key ID, questo campo viene posto a 0.

* **QKD KEY RESPONSE**

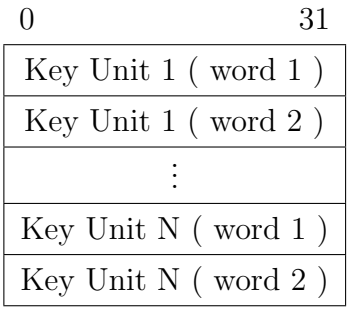


Figura 3.9: QKD KEY REQUEST

Viene inviato come risposta al QKD KEY REQUEST, dal QKD device verso il Gateway IPsec. Esso contiene un Numero N di Key Unit, dove N è pari al valore Num Key Unit inviato in precedenza. Quando non è possibile reperire nuove chiavi, viene inviato una sola Key Unit pari a 0 per segnalare l'errore.

3.2.4 Generazione chiavi per le IKE SA e CHILD SA

Le scelte che sono state effettuate consistono nell'utilizzare la QKD in due modalità:

- I bit prodotti dalla QKD vengono utilizzati per generare le chiavi per tutti i servizi, eccetto la componente SK_d. Ogni volta che risulterà necessario generare nuove chiavi, queste verranno quindi prodotte a partire da bit sempre nuovi inviati dal QKD device.

Algoritmo 4 Generazione chiavi per le IKE SA usando la QKD

Definiamo
$\|$: l'operazione di concatenazione.
SK_{ei} e SK_{er}: chiavi usate per la crittazione.
SK_{ai} e SK_{ar}: chiavi usate per il controllo d'integrità.
SK_{pi} e SK_{pr}: chiavi usate per generare il payload AUTH.
keybuffer: bit provenienti dal QKD device.

$SK_d \| SK_{ai} \| SK_{ar} \| SK_{ei} \| SK_{er} \| SK_{pi} \| SK_{pr} =$ keyBuffer

L'algoritmo per generare le chiavi per le CHILD SA è lo stesso con l'unica differenza che non vengono generate le componenti SK_{pi} e SK_{pr}, come prevede anche IKEv2.

- I bit prodotti dalla QKD vengono utilizzati allo stesso modo di un master secret DH. A parte questo, il funzionamento è lo stesso visto con IKEv2.

3.2.5 Considerazioni sulle Fallback

Lo scenario descritto da Nagayama e Van Meter considera il QKD device come descritto nel secondo capitolo, e infatti è previsto che solamente l'initiator, questa volta inteso come chi inizia una richiesta di rekeying e quindi non necessariamente l'initiator "assoluto", possa riscontrare dei problemi nel reperimento della chiave. Nella comunicazione tramite QKD in mancanza di componenti aggiuntive quali ad esempio server o database, per la simmetria che la caratterizza non è possibile che chi riceve il messaggio di rekeying non sia in grado di ottenere una chiave QKD mentre chi ha inviato il messaggio sì.
Considerando invece la realtà che abbiamo descritto, questo non è sempre verificato in quanto in ogni QKD device è presente, oltre al sistema QKD, un database e un QKD server, entrambi soggetti a possibili malfunzionamenti che possono accadere indipendentemente su ogni device.
Nel caso il metodo di Fallback scelto sia WAIT QKD o CONTINUE, non sussiste alcun problema, in quanto il responder, inteso come chi risponde ad una richiesta di rekeying, non dovrebbe fare altro che inviare nella risposta il payload QKD KEY ID con i campi Key ID e No Key rispettivamente a 0 e 1. Ora, a seconda

della situazione in cui ci troviamo, l'initiator deve rispettivamente, aspettare che il responder sia in grado di ottenere la chiave dal proprio QKD device, o continuare ad utilizzare la chiave più recentemente utilizzata.
Nel caso invece in cui il metodo di Fallback scelto sia DIFFIE HELLMAN, il responder non sarebbe in grado di generare le chiavi non avendo ricevuto il payload KE dell'initiator (non viene inviato da quest'ultimo, non avendo riscontrato problemi durante il reperimento della chiave). La soluzione che è stata scelta consiste nell'inviare sempre i payload KEi e KEr, quando viene scelto questo metodo di Fallback, in modo che in qualunque situazione entrambe le parti siano in grado di generare le chiavi DH correttamente. Nonostante questo causi un piccolo overhead, siamo in grado di risolvere il problema in una maniera molto semplice da implementare e che non modifica IKEv2 più di quanto non sia già stato descritto in precedenza.

In tutti questi casi, quando scade il lifetime della SA o il timeout impostato per il metodo WAIT QKD, se il QKD device non riesce nuovamente a fornire i bit necessari, la comunicazione viene interrotta. E' importante inoltre che il metodo di Fallback sia scelto all'inizio della comunicazione e che non possa essere negoziato nuovamente in seguito, per cercare ad esempio di adattarsi a eventuali malfunzionamenti. Questo da una parte fornisce una certa rigidità alla nostra architettura, ma dall'altra risulta necessario per garantire uno standard di sicurezza elevato.

Capitolo 4

Implementazione

La maggior parte delle implementazioni di IPsec consiste in due moduli comunicanti, uno che realizzi ESP, AH e il packet processing che operi a livello kernel e un altro che realizzi IKE che operi invece a livello utente.

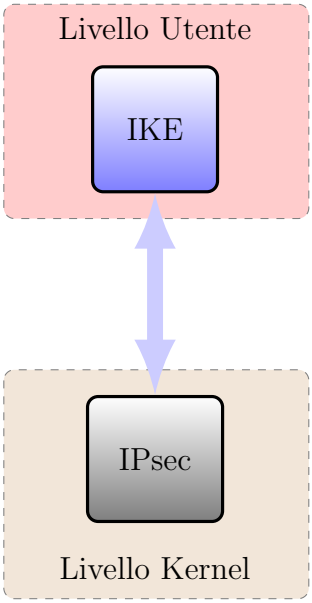


Figura 4.1: Implementazione generica di IPsec e IKE

Considerato che la quasi totalità dei cambiamenti derivanti dalla nostra integrazione sono state fatte su IKEv2, è stato scelto di utilizzare e modificare una implementazione per IKEv2 già esistente, OpenIKEv2[12]. Verrà quindi data una panoramica delle sue funzionalità e successivamente verranno descritte le modifiche che sono state apportate per integrare l'architettura per la QKD descritta nel precedente capitolo.

4.1 OpenIKEv2

OpenIKEv2 è un'implementazione open-source di IKEv2 scritta in C++, sviluppata dai ricercatori del gruppo di ricerca ANTS[1] dell'università di Murcia. Il suo codice è stato interamente scritto secondo i paradigmi dell'Object Oriented Programming, il che ha permesso anche a noi di lavorare più agevolmente. Come interfacce per IPsec possono essere utilizzate sia `XFRM`[2] sia `PFKEYv2`[5]. Inoltre presenta la possibilità di configurare le policy e altre preferenze senza dover eseguire comandi specifici quali ad esempio `setkey`[3], ma semplicemente utilizzando un file di configurazione.
La sua struttura è divisa in 3 componenti principali: libopenikev2, libopenikev2_impl e openikev2.

4.1.1 libopenikev2

Il componente libopenikev2 consiste in una libreria in cui sono definite le classi e interfacce su cui basarsi per gestire la quasi totalità dei servizi necessari ad IKEv2, come ad esempio la generazione di chiavi DH per le IKE SA e CHILD SA, vari controller per le IKE SA e IPsec, la gestione degli eventi e della rete ecc. Le classi più importanti sono le seguenti:

IkeSa: la classe IkeSa realizza una IKE SA. Essa contiene tutti i metodi necessari per la creazione dei messaggi caratteristici di IKEv2.

Payload_ke, Payload_auth ecc: tutte le classi denominate come payload_*nome_payload* rappresentano ogni singolo payload definito in IKEv2. Ognuna di queste classi presenta dei metodi comuni necessari per le operazioni di base sui pacchetti. Questi metodi sono i seguenti:

* **getBinaryRepresentation:** aggiunge la rappresentazione in bit del payload ad un buffer passato come input.
* **parse:** crea una nuova istanza della classe a rappresentante il payload, a partire dalla sua rappresentazione in bit.
* **clone:** crea una copia di un'istanza del payload.
* **toStringTab:** permette di stampare il payload in modo che ogni campo sia ben definito e comprensibile. Ad esempio, il payload SA verrebbe stampato nel seguente modo:

[1] Vedi http://ants.dif.um.es/
[2] Si veda il codice del kernel Linux http://www.kernel.org/
[3] Vedi http://linux.die.net/man/8/setkey

```
<PAYLOAD_SA> {
                  <PROPOSAL> {
                     proposal_number=1
                     protocol_id=PROTO_IKE
                     spi=[]
                     <TRANSFORM> {
                         type=ENCR
                         id=ENCR_3DES
                     }
                     <TRANSFORM> {
                         type=ENCR
                         id=ENCR_DES
                     }
                     <TRANSFORM> {
                         type=INTEG
                         id=AUTH_HMAC_SHA1_96
                     }
                     <TRANSFORM> {
                         type=INTEG
                         id=AUTH_HMAC_MD5_96
                     }
                     <TRANSFORM> {
                         type=PRF
                         id=PRF_HMAC_MD5
                     }
                     <TRANSFORM> {
                         type=DH
                         id=2
                     }
                     <TRANSFORM> {
                         type=DH
                         id=1
                     }
                  }
               }
```

ChildSa: realizza il concetto di CHILD SA.

4.1.2 libopenikev2_impl

In questo componente vengono definite tutte le implementazioni che realizzano le interfacce definite in libopenikev2. Fra tutte le classi presenti, le più importanti sono le seguenti:

* **Authenticator:** questa classe realizza l'interfaccia authenticator per fornire il servizio di autenticazione basato su chiave precondivisa o su certificati di qualunque tipologia. E' inoltre possibile utilizzare il protocollo EAP.

* **IpsecControllerImpl:** questa interfaccia viene realizzata da due classi che utilizzano rispettivamente un socket netlink (per `XFRM`) e uno `PFKEYv2`.

* **NetworkControllerImpl, IpAddress:** realizzano le interfacce per gestire la rete utilizzando principalmente la libreria standard per i socket. E' possibile la risoluzione di un indirizzo IP tramite server DHCP.

* **ThreadControllerImpl, Mutex, Condition:** realizzano le interfacce necessarie alla gestione dei thread. A questo scopo viene utilizzata principalmente la libreria pthread.

* **CryptoController, Cipher, DH, Random, KeyRing:** queste interfacce vengono implementate servendosi della libreria `libcrypto` presente nel progetto openssl. Esse permettono di utilizzare i più famosi algoritmi crittografici quali ad esempio AES, 3DES, SHA1, MD5 e i gruppi MODP[4] 1, 2, 5, 14, 15, 16, 17 e 18.

* **LogImpl:** implementa il Log di sistema. Sono presenti tre versioni del Log in modo tale da stampare i dati su un file di testo colorato o meno e come codice html.

4.1.3 openikev2

openikev2 è l'ultimo componente e realizza l'applicazione vera e propria che viene eseguita, utilizzando le classi e interfacce definite negli altri due componenti. Esso funziona come un normale demone[5] Unix. I file più importanti di cui è composto sono i seguenti:

* **openikev2.cpp:** realizza il programma principale. Ha il compito di inizializzare tutti i sotto-sistemi che operano in background quali i controller dei thread, di IPsec, delle IKE SA ecc.

* **configurerlibconfuse.cpp:** questa classe utilizza la libreria libconfuse[6] per fare il parsing del file di configurazione.

4.2 Modifiche apportate

Verranno ora descritte tutte le modifiche, classe per classe, apportate per integrare la QKD e realizzare la parte dell'architettura descritta in precedenza compresa nel Gateway IPsec.

4.2.1 IkeSa

Per ovvi motivi, la classe IkeSa è quella che è stata modificata maggiormente, in quanto al suo interno sono descritti tutti gli scambi di messaggi visti nel primo capitolo. Ogni metodo che si occupa di creare o elaborare un messaggio in uscita

[4] Vedi http://tools.ietf.org/html/rfc3526
[5] Programma eseguito in background senza che sia sotto il controllo diretto dell'utente
[6] http://www.nongnu.org/confuse/

o in arrivo, presenta un nome molto chiaro che ci aiuta a capire di cosa si occupa. A titolo di esempio il metodo createIkeSaNegotiationRequest crea e invia il messaggio IKE SA INIT request.
Sono stati quindi modificati tutti i metodi che si occupano degli scambi IKE SA INIT, IKE SA AUTH e CREATE CHILD SA, modificando i payload inviati secondo quanto descritto nel capitolo 3.
In particolare, la maggior parte delle modifiche è stata effettuata sui metodi createIkeSaNegotiationRequest, processIkeSaNegotiationRequest, createIkeSaNegotiationResponse e processIkeSaNegotiationResponse che vengono chiamati dai metodi principali per creare ed elaborare i payload SA,KE e N (nonce) e richiamare i metodi che si occupano di creare il master secret DH e le chiavi per le IKE SA. La nuova versione di questi metodi, in accordo con quanto scritto nel precedente capitolo, invia ed elabora sempre i payload SA e QKD KEY ID, e aggiunge ed elabora il nonce quando vengono chiamati per operazioni di rekeying. Inoltre vengono create le chiavi per le IKE SA, utilizzando la QKD, chiamando dei nuovi metodi che verranno descritti in seguito. Lo stesso è stato effettuato sui metodi relativi alle CHILD SA che portano lo stesso nome dei metodi appena descritti, con l'unica differenza che è scritto ChildSa invece che IkeSa.

La gestione delle Fallback avviene sempre in questi metodi in maniera molto semplice. Tramite il costrutto switch case, in base a quale metodo di fallback è stato lanciato, viene svolta l'operazione dedicata.

Per quanto riguarda l'IKE SA AUTH, sono stati modificati i metodi relativi che ora inviano ed elaborano il payload QKD FALLBACK e richiamano degli altri nuovi metodi descritti in seguito per effettuare l'autenticazione senza utilizzare il nonce.

4.2.2 Payload QKD KEY ID

E' stata creata una nuova classe che realizza il Payload QKD KEY ID, descritto nel precedente capitolo. La struttura della classe è sostanzialmente la stessa degli altri payload e presenta quindi gli stessi metodi comuni. Le variabili di classe rappresentano ogni campo del payload e sono state definite nel seguente modo:

```
1 public:
2
3     uint8_t version;
4     uint noKeyBit : 1;
5     uint flags : 7;
6     uint32_t keyID;
```

Come identificativo del payload, in accordo con quanto scritto nella sez. 3.2 di [5], è stato assegnato il valore 128, utilizzabile per scopi non specificati nello standard.

4.2.3 Payload QKD FALLBACK

E' stata creata una nuova classe che realizza il Payload QKD FALLBACK, descritto nel precedente capitolo. Anche in questo caso, la struttura della classe è sostanzialmente la stessa degli altri payload e presenta quindi gli stessi metodi comuni. Le variabili di classe rappresentano ogni campo del payload e sono state definite nel seguente modo:

```
public:

 enum QKD_FALLBACK_METHOD
 {
  WAIT_QKD = 1,
  DIFFIE_HELLMAN,
  CONTINUE,
 };

 uint8_t version;
 uint8_t FLAGS;
 QKD_FALLBACK_METHOD fallback;
```

Come identificativo del payload, in accordo con quanto scritto nella sez. 3.2 di [5], è stato assegnato il valore 129, utilizzabile per scopi non specificati nello standard.

4.2.4 TcpSocket

E' stata aggiunta una classe che implementasse un socket TCP, utilizzando la libreria standard. I metodi principali sono send e rcv che rispettivamente inviano e ricevono dati tramite il socket creato.

4.2.5 QKDHandler

La classe QKDHandler è stata aggiunta al componente libopenikev2_impl per gestire la comunicazione fra Gateway IPsec e QKD device, dal lato del Gateway. Essa presenta solamente il metodo statico getQKDkey che, a partire dal Key ID (ID) e dal numero di Key Unit richieste (numIDs), invia il payload QKD KEY REQUEST al QKD device e attende la risposta di quest'ultimo.

```
auto_ptr<ByteArray> QKDHandler::getQKDkey(uint32_t ID, uint32_t numIDs) {

    try {
      // Establish connection with the server
      TCPSocket sock(SERVADDRESS, SERVPORT);

      // Create the packet containing the request for the QKD subsystem
      ByteBuffer packetToSend (SENDPACKETSIZE);
      packetToSend.writeInt32(ID);
      packetToSend.writeInt32(numIDs);

      // Send the packet to the QKD subsystem
      sock.send(packetToSend.getRawPointer(), packetToSend.size());

```

```
15      uint keyBufferLength;
16      ByteBuffer keyBuffer (0);
17
18      if(numIDs < (UINT_MAX / KEYUNITLENGTH)){
19         keyBufferLength = KEYUNITLENGTH * numIDs;
20         keyBuffer.setSize(keyBufferLength);
21      }
22      else{
23        Log::writeLockedMessage("<QKDHANDLER>:","Too many Key Units",Log::
              LOG_ERRO,true);
24        return auto_ptr<ByteArray> (new ByteArray(0));
25      }
26
27      // Bytes read on each recv()
28      uint bytesReceived = 0;
29
30      // Receive up to the buffer size from the sender
31      uint8_t* tempBuffer = keyBuffer.getRawPointer();
32      if ((bytesReceived = (sock.recv(tempBuffer, keyBufferLength))) <= 0) {
33        Log::writeLockedMessage("<QKDHANDLER>:","Unable to read the socket
              buffer",Log::LOG_ERRO,true);
34        return auto_ptr<ByteArray> (new ByteArray(0));
35      }
36      keyBuffer.writeBuffer(tempBuffer,bytesReceived);
37      Log::writeLockedMessage("<QKDHANDLER>:","key material sent from QKD
            subsystem" + keyBuffer.toStringTab(1),Log::LOG_CRYP,false);
38      return keyBuffer.clone();
39
40
41    } catch(NetworkException &e) {
42      Log::writeLockedMessage("<QKDHANDLER>:","Error trying to connect to our
            QKD system",Log::LOG_ERRO,true);
43      return auto_ptr<ByteArray> (new ByteArray(0));
44    }
45
46
47  }
```

4.2.6 KeyRingQKD

E' stata creata una nuova implementazione dell'interfaccia KeyRing, KeyRingQKD che comprende una nuova versione dei metodi generateIkeSaKeys e generateChildSaKeys che generano rispettivamente le chiavi per le IKE SA e CHILD SA con la QKD secondo la modalità viste nel capitolo 3.

```
1  int KeyRingQkd::generateIkeSaKeys( uint32_t* keyID , bool useLastKey) {
2
3      auto_ptr<ByteArray> temp;
4
5      if(useLastKey){
6        //only used with the CONTINUE fallback method
7        temp = KeyRingQkd::lastKeyBufferSA->clone();
8      }
9      else{
10       uint32_t total_size = this->integ_key_size*2 + this->encr_key_size*2 +
             this->prf->prf_size*2;
11
12       uint32_t numIDs = ceil(((float) total_size / (float) QKDHandler::
             KEYUNITLENGTH));
13
```

```
14      temp = QKDHandler::getQKDkey(*keyID, numIDs );
15
16      if(temp->size() ==0 || temp->operator ==(0)){
17        Log::writeLockedMessage("<KEYRINGQKD:>","Server is down or there is no
              available key",Log::LOG_ERRO,true);
18        return EXIT_FAILURE;
19      }
20
21      *keyID += numIDs;
22
23      //we store the last used buffer for future uses
24      KeyRingQkd::lastKeyBufferSA = temp->clone();
25
26
27    }
28    ByteBuffer buffer (*temp);
29    // Copies the values of the keys on their buffers
30    this->sk_ai = buffer.readByteArray( this->integ_key_size );
31    this->sk_ar = buffer.readByteArray( this->integ_key_size );
32    this->sk_ei = buffer.readByteArray( this->encr_key_size );
33    this->sk_er = buffer.readByteArray( this->encr_key_size );
34    this->sk_pi = buffer.readByteArray( this->prf->prf_size );
35    this->sk_pr = buffer.readByteArray( this->prf->prf_size );
36
37    return EXIT_SUCCESS;
38  }
```

```
1   int KeyRingQkd::generateChildSaKeys( uint32_t* keyID, bool useLastKey ){
2
3     auto_ptr<ByteArray> temp;
4     //only used with the CONTINUE fallback method
5     if(useLastKey){
6       temp = KeyRingQkd::lastKeyBufferChildSA->clone();
7     }
8     else{
9       uint32_t total_size = ( this->encr_key_size + this->integ_key_size ) * 2;
10
11      uint32_t numIDs = ceil(((float) total_size / (float) QKDHandler::
            KEYUNITLENGTH));
12
13      temp = QKDHandler::getQKDkey(*keyID, numIDs );
14
15      if(temp->size() ==0 || temp->operator ==(0)){
16        return EXIT_FAILURE;
17      }
18
19      *keyID += numIDs;
20
21      KeyRingQkd::lastKeyBufferChildSA = temp->clone();
22
23
24    }
25
26    ByteBuffer buffer (*temp);
27
28    // Copies the values of the keys on their buffers
29    this->sk_ei = buffer.readByteArray( this->encr_key_size );
30    this->sk_ai = buffer.readByteArray( this->integ_key_size );
31    this->sk_er = buffer.readByteArray( this->encr_key_size );
32    this->sk_ar = buffer.readByteArray( this->integ_key_size );
33
34    return EXIT_SUCCESS;
35  }
```

4.2.7 Authenticator

Sono stati aggiunti dei metodi, generateAuthDataNoNonceToBeSigned, generatePskAuthNoNonceField e che permettono di creare il payload AUTH senza l'utilizzo dei nonce, in accordo con quanto scritto nel capitolo 3. E' stato in seguito modificato il metodo verifyAuthPayload in modo da potersi autenticare anche con questa nuova modalità.

4.3 QKD device utilizzato per i test

4.3.1 Sistema QKD

La generazione delle chiavi da parte di sistema QKD reale, è stata simulata tramite un generatore pseudo-aleatorio. Per fare in modo che vengano generate le stesse chiavi su entrambi i QKD device è stato usato lo stesso seed per ogni generatore.
Il valore di ogni Key Unit viene scelto come sottoinsieme random di un charset definito a priori, nel nostro caso pari a,

`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.`

La sua implementazione è definita nella classe QKeyGenerator, il cui metodo principale per generare una singola chiave è il seguente

```
string QKeyGenerator::generateKey(){
     string key = "";
     for(int i = 0; i < this->keylength; i++){
       key += QKeyGenerator::charset()[rand() % 62];
     }
     return key;
   }
```

4.3.2 Database

Per implementare il database si è utilizzato il DBMS MySQL[7]. E' stato creato un database di nome QKDKEYS la cui unica tabella realizza quella descritta nel capitolo 3 e il cui codice SQL è il seguente:

```
CREATE TABLE IF NOT EXISTS `KEYS` (
  `time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
      CURRENT_TIMESTAMP,
  `keyid` int(11) NOT NULL,
  `keyunitvalue` varchar(8) CHARACTER SET utf8 NOT NULL,
  `flag` tinyint(1) NOT NULL,
  PRIMARY KEY (`keyid`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

[7] Vedi http://www.mysql.it/

4.3.3 QKD server

Il QKD server è stato implementato come un server TCP canonico. Per realizzarlo è stata usata la classe TCPSocket vista in precedenza. Il server compie ciclicamente i seguenti passi:

1. Attende un QKD KEY REQUEST dal Gateway IPsec.
2. Una volta ricevuto, traduce la richiesta in una query SQL, in questo modo,

```
1 //we build the query. It has to be a const char*
2 stringstream sqlstring;
3 sqlstring << "SELECT `keyunitvalue` FROM `KEYS` WHERE `keyid`= ";
4 sqlstring << keyID;
5 const char* sqlcommand = sqlstring.str().c_str();
6
7 //we make the query to the chosen DB server
8 query_state = mysql_query(connection, sqlcommand);
```

   che viene ripetuta un numero di volte pari al numero di Key Unit richieste.
3. Invia al Gateway IPsec il payload QKD KEY RESPONSE, contenente le Key Unit richieste.

4.4 Considerazioni finali

Ora che abbiamo a disposizione un'implementazione di IPsec integrato con la QKD, ci restano da definire le modalità con cui determinare i valori ottimali per i parametri configurabili dall'utente o amministratore di sistema. Infatti la scelta di un valore rispetto ad un altro può determinare dei miglioramenti o peggioramenti sostanziali per quanto concerne il grado di sicurezza, efficienza e affidabilità del nostro sistema.
Le domande principali che dobbiamo porci sono le seguenti:

+ *Che metodo di fallback scegliere?*
+ *Qual è il valore opportuno come lifetime (reale) di una chiave?*
+ *Qual è il numero di Key Unit minimo che deve contenere il nostro DB?*
+ *Qual è la combinazione di algoritmi che garantisce il grado più alto di sicurezza e efficienza?*

In tutti i casi, con "opportuno" si intende che riduce la probabilità che venga lanciato un metodo di Fallback sotto una soglia auspicabile.

Rispondere alla prima domanda non è semplice, e non vi è una risposta assoluta in quanto questa dipende dalle potenzialità del sistema QKD che abbiamo, in

particolare il rate con cui vengono prodotte le chiavi, e dalle necessità dell'utente. Infatti sulla carta, il metodo WAIT QKD risulterebbe il più sicuro in quanto non riutilizza nessuna chiave nè genera le chiavi con DH, però se il rate del nostro sistema QKD è molto basso e/o il timeout da noi impostato è alto, saremmo soggetti a numerose attese e la possibilità che la comunicazione venga interrotta aumenterebbe di molto. D'altra parte, se utilizzassimo invece CONTINUE o DIFFIE-HELLMAN non dovremmo necessariamente attendere il sistema QKD, ma peggioreremmo il grado di sicurezza della nostra comunicazione e un utente potrebbe preferire di bloccare la comunicazione piuttosto che questo accada.

Per rispondere alle altre domande possiamo ragionare nel seguente modo. Chiamando L il lifetime per le SA impostato dall'utente, per evitare che due utenti che impostano lo stesso valore lancino una richiesta di rekeying in simultanea, esso viene modificato da OpenIKEv2 in modo random. Il valore minimo reale che può essere effettivamente utilizzato risulta essere:

$$L_{realmin} = 90\%L.$$

Per far sì che non venga mai lanciato un metodo di fallback è necessario, a meno di malfunzionamenti, che ogni volta che vengono richieste nuove Key Unit, il numero di quelle presenti sul database sia maggiore o uguale a quello richiesto. Trascurando i tempi per l'inserimento della chiave nel DB, possiamo definire il numero di Key Unit producibili dalla QKD e memorizzabili in un intervallo di tempo Δt come

$$N_{keyUnitDB\Delta t} = \left\lfloor \frac{R_{QKD} \cdot \Delta t}{L_{keyUnit}} \right\rfloor.$$

Concordemente a quanto è stato detto, questo valore deve verificare la seguente disequazione ($\Delta t = L_{realmin}$):

$$\text{Numero Key Unit richieste} \leq \left\lfloor \frac{R_{QKD} \cdot L_{realmin}}{L_{keyUnit}} \right\rfloor,$$

dove R_{QKD} è il rate con cui il nostro sistema QKD genera nuove chiavi e $L_{keyUnit}$ è il valore scelto come lunghezza per le Key Unit. Si nota quindi, come l'unico valore non arbitrario fra questi sia R_{QKD}. Considerando che allo stato attuale questo valore è basso, per configurare correttamente il nostro sistema, bisognerà trovare il giusto compromesso fra gli altri valori, in modo che la disequazione sia verificata, cercando inoltre di limitare il più possibile il lifetime della chiave per garantire uno standard elevato di sicurezza e di non utilizzare Key Unit troppo corte per non effettuare troppi accessi al DB.

Capitolo 5

Sviluppi futuri

Come descritto nel precedente capitolo, allo stato attuale, non è stato possibile testare la nostra implementazione con un sistema QKD reale. E' necessario quindi, che in futuro la si testi con una strumentazione adeguata e eventualmente la si modifichi. Inoltre, come si può notare, una prima critica che si potrebbe effettuare al nostro lavoro è che non sfrutta completamente le potenzialità offerte dalla QKD in quanto provengono sì da essa, ma per crittare i dati si utilizza sempre e comunque algoritmi come AES, 3DES ecc. Quindi, anche se la nostra chiave risulta essere perfettamente sicura, siamo comunque soggetti ad eventuali debolezze derivanti dall'utilizzo di questi algoritmi.
Allo scopo di migliorare questa situazione, si potrebbe pensare di aggiungere un algoritmo one time pad[1] usando le chiavi QKD. In questo modo si sfrutterebbe la QKD nella sua completezza e si riuscirebbe a fornire una sicurezza teoricamente incondizionata.
Un altro possibile spunto per un'ulteriore analisi potrebbe essere valutare la fattibilità, considerando il canale fra Gateway IPsec e QKD device perfettamente sicuro, di effettuare l'autenticazione fra le due parti tramite QKD[2].

[1] Algoritmo crittografico che utilizza chiavi monouso e lunghe quanto la porzione di dati da cifrare.

[2] Si veda ad esempio il lavoro dei ricercatori Guihua Zeng e Xinmei Wang, esposto nell'articolo *"Quantum key distribution with authentication"*, reperibile su questo sito http://arxiv.org/abs/quant-ph/9812022

Conclusioni

In questa tesi, si è potuto constatare come sia spesso complicato occuparsi di sicurezza nel campo delle reti cercando da una parte di garantire un livello di protezione elevato e dall'altra un certo grado di efficienza. Nella maggior parte queste due caratteristiche si rivelano essere contrastanti ed è quindi stato necessario trovare il giusto compromesso fra efficienza e sicurezza.
In particolare, nel nostro caso, il problema maggiore è stato definire un'architettura che complicasse il meno possibile la già intricata e granulare struttura di IPsec vista e criticata nel primo capitolo, considerando anche i problemi di sincronizzazione derivanti dall'introduzione della Quantum Key Distribution in esso.
Successivamente si è dovuto trovare un modo per implementare questa architettura e per testarla senza avere la possibilità di usufruire di un sistema QKD reale, e di cercare di effettuare comunque delle valutazioni sulla fattibilità dell'integrazione.
Si può prevedere comunque, che quando l'architettura e la versione modificata di OpenIKEv2 verranno utilizzate insieme ad un sistema QKD reale, saranno necessari alcuni aggiustamenti e modifiche per migliorare l'efficienza e la flessibilità della struttura.
In ogni caso questo lavoro servirà da base per sviluppare un'applicazione reale della QKD e che possa portare un'ulteriore contributo a chi si occupa di sicurezza delle reti e in particolare di integrare la Quantum Key Distribution con i protocolli già esistenti.

Elenco delle figure

Bibliografia

[1] **S. Kent, K. Seo**. *Security Architecture for the Internet Protocol*. IETF. Dicembre 2005. RFC 4301.

[2] **S. Friedl**. *An illustrated Guide to IPsec*. `http://www.unixwiz.net/techtips/iguide-ipsec.html`. Agosto 2005.

[3] **S. Kent**. *IP Authentication Header*. IETF. Dicembre 2005. RFC 4302.

[4] **S. Kent**. *IP Encapsulating Security Payload (ESP)*. IETF. Dicembre 2005. RFC 4303.

[5] **C. Kaufman, P. Hoffman, Y. Nir e P. Eronen**. *Internet Key Exchange version 2 (IKEv2)*. IETF. Settembre 2010. RFC 5996.

[6] **N. Ferguson, B. Schneier**. *A Cryptographic Evaluation of IPsec*. Febbraio 1999.

[7] **K. G. Paterson, Arnold K. L. Yau**. *"Cryptography in Theory and Practice: The Case of Encryption in IPsec"*. [ed.] Vaudenay, S. Springer-Verlag, 2006. Vol. Lecture Notes in Computer Science Vol. 4004.9783540345466.

[8] **S. M. Bellovin**. *Problem Areas for the IP Security Protocols*. San Jose, CA, July 1996.

[9] **C. H. Bennet, G. Brassard** *"Quantum cryptography: Public key distribution and coin tossing"*. *Proceedings of IEEE International Conference on Computers Systems and Signal Processing*, Vol. 175, pp. 175-179, Bangalore, India, 1984.

[10] **C. H. Bennett, F. Bessette, G. Brassard, L. Salvail, e J. Smolin**. *Experimental quantum cryptography*, J. Cryptol. 5, 1, 3-28. Gennaio 1992.

[11] **S. Nagayama, R. Van Meter**. *IKE for IPsec with QKD*. Internet Draft. IETF. October 19, 2009.

[12] **A. Pérez Méndez, P. J. Fernández Ruiz, G. Martínez Pérez, R. M. López, A. F. Gómez Skarmeta**, *Documentazione del software* OpenIKEv2, `http://openikev2.sourceforge.net/index.html`.